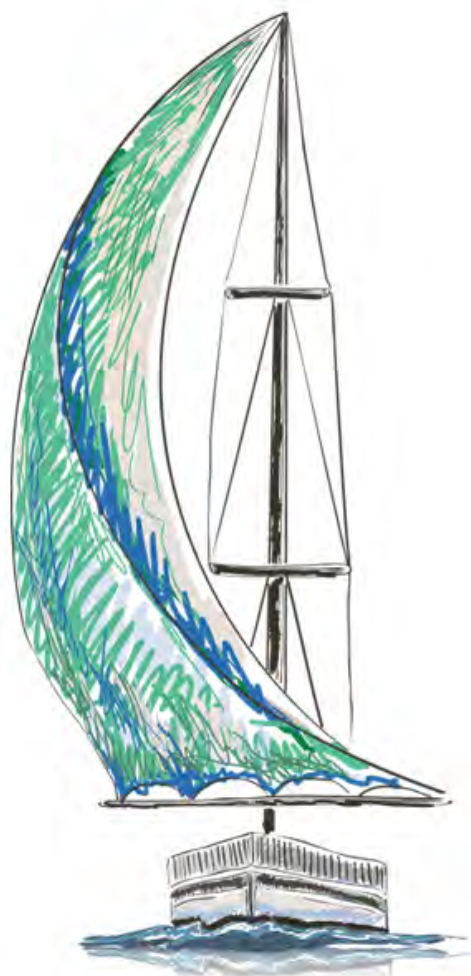
Falando de Grails

Altíssima produtividade no desenvolvimento web

HENRIQUE LOBO WEISSMANN

Sumário

Capítulo 1

Expulsando o tédio da plataforma Java EE

Não me surpreenderia se você estivesse buscando saber mais a respeito de Groovy e Grails devido a um dos motivos a seguir:

- Aquela impressão de que o modo como trabalhamos com Java e Java EE poderia ser mais pragmático, menos burocrático e mais produtivo. Talvez até mesmo tão produtivo quanto sua plataforma de desenvolvimento atual.
- Aquele desejo de ser mais produtivo com um custo menor.
- Você gostaria de entrar para o "mundo Java" mas se sente inseguro com a quantidade de siglas e nomes que normalmente envolvem esta plataforma. Nomes como EJB, JNDI, Servlet, JPA, JAX-RS, JAX-WS e tantos

outros. Parece uma infinidade de tecnologias que, se você não dominar, vai conseguir no máximo resultados medíocres.

- Você não acredita que seja necessário conhecer tantas tecnologias para "apenas escrever uma aplicação web simples".
- Simplesmente ouviu falar sobre Groovy e Grails e ficou curioso a respeito.

Infelizmente, mesmo após tantos avanços, a plataforma Java EE ainda carrega uma imagem negativa para muitos desenvolvedores. Nunca conheci alguém que negasse o seu poder, mas sempre me deparo com aqueles que a veem como um mundo inacessível, excessivamente complexo e burocrático. E devo confessar, essas pessoas estão certas e Grails prova isto.

Grails nos apresenta uma maneira diferente de manipularmos a plataforma Java EE, mais simples, menos burocrática e extremamente produtiva. E sabem o que é mais interessante? Um modo de trabalho no qual o desenvolvedor simplesmente se esquece de estar lidando com o Java EE: você simplesmente a usa.

Este "novo modo de trabalho" na realidade é consequência da linguagem de programação na qual Grails é baseado, Groovy, que, na minha opinião, é onde o tesouro realmente se encontra. Groovy foi criado com o objetivo de facilitar o desenvolvimento de aplicações para a JVM. Podemos dizer que resolve dois problemas fundamentais: ser compatível com o código Java existente, garantindo completo reaproveitamento de código e, ao mesmo tempo, resolver diversas das limitações existentes na própria linguagem Java.

Como o leitor verá no decorrer deste livro, não é exagero dizer que Groovy é como um "Java++": diversos novos recursos que só foram incluídos no Java 8 em 2014 já estavam presentes em Groovy desde sua introdução em 2004 como, por exemplo, *closures*, facilidades para lidar com tipos numéricos e diversas outras facilidades sintáticas que aumentam significativamente a produtividade do programador.

Enquanto Groovy facilita o trabalho do programador, eliminando as dificuldades inerentes da linguagem Java, o Grails torna acessível todos os recursos da plataforma Java EE de uma forma nunca antes vista para aqueles que

desejam tirar proveito de suas principais vantagens: alta disponibilidade, escalabilidade, desempenho e robustez. Não há como negar, o desenvolvimento de aplicações web na plataforma Java muitas vezes é uma atividade mais burocrática que o necessário. Grails remove este aspecto moroso ao nos oferecer um modelo de desenvolvimento baseado em convenções: arquivos de configuração agora somente quando for o desejo do programador, e não mais uma exigência da plataforma.

Groovy e Grails acabam com o tédio que pode se tornar trabalhar com a plataforma Java EE. E finalizado este livro o leitor poderá aproveitar todos os seus recursos de uma forma surpreendentemente simples e direta.

1.1 Groovy e Grails são pontes

Na minha experiência, o maior ganho destas tecnologias são o fato de serem "pontes". Quando uso esta metáfora, refiro-me ao fato de elas nos possibilitarem reaproveitar conhecimentos que já possuímos e muitas vezes sequer nos damos conta ou valorizamos.

A primeira grande ponte na minha opinião é a linguagem Groovy. Não é raro que meus clientes sejam desenvolvedores proficientes em tecnologias que ficam fora do ambiente Java e posteriormente passem a dominá-lo justamente por terem usado como ponte o Groovy.

Groovy é uma linguagem cuja sintaxe é fácil de ser entendida por aqueles que não estão acostumados ao Java. Coisas simples como o caractere `;` ou mesmo o uso de parênteses em chamadas de funções podem parecer burocráticos para estes programadores (e eles estão certos). Groovy os torna opcionais, assim como tantas outras "frescuras", o que acaba gerando uma porta de entrada ao ecossistema Java muito mais hospitaleira.

O contrário também é verdade: vejo programadores Java se tornarem melhores programadores Java graças ao Groovy (é o meu caso). Você tem a liberdade de ir experimentando aos poucos conceitos que muitas vezes parecem assustadores, como tipagem dinâmica, programação funcional e tantos outros. E sabe o que é mais legal? A sintaxe do Groovy é próxima à do Java: na maior parte das vezes se você compilar seu código Java com o compilador Groovy, não há problemas.

Mas como assim? No penúltimo parágrafo você disse que a sintaxe do Groovy é parecida com a de linguagens de programação que estão fora do ecossistema Java, e no último falou que parece com a do Java. Como assim??? Isto mesmo: Groovy é próxima dos dois extremos e você o verá com detalhes a partir dos próximos capítulos.

Voltando à ponte, você não precisa dar grandes saltos, você caminha normalmente de uma extremidade a outra aproveitando o que você já conhece e aprendendo novas coisas a cada passo. Groovy me ensinou programação poliglota, o que uma linguagem dinâmica realmente é, programação funcional, novas maneiras de se pensar AOP e mesmo me forneceu uma visão mais crítica em relação ao Java e diversas outras linguagens. E tudo isto reaproveitando o que já conhecia: meu conhecimento sobre o Java e as outras linguagens que me formaram (PHP, Delphi, Visual Basic).

Grails é a ponte que nos leva a dominar a plataforma Java EE. Não é um exagero dizer que você não precisa saber praticamente nada sobre o Java EE para desenvolver uma aplicação web com Grails. Pouco a pouco você irá experimentando mais o framework, curioso a respeito do seu funcionamento e, conforme "caminha por esta ponte", um belo dia perceberá que agora conhece uma série (talvez todas) daquelas siglas e nomes que soavam estranhos em seu primeiro contato com o Java EE.

E se você for um programador que já domina o Java EE? Grails lhe fornecerá uma visão mais crítica a respeito do modo como você interagia com suas tecnologias e, não raro, inclusive lhe apresentará em alguns momentos a aspectos da plataforma com os quais, muitas vezes, acidentalmente irá conhecer neste caminho.

1.2 Por que Groovy?

Existe uma certa resistência ao Groovy por ser visto como uma "mera linguagem de script" e que, portanto, não seria apropriada para a escrita de aplicações reais. Nada mais longe da verdade: Groovy tem sido usado em projetos gigantescos de imensa responsabilidade por todo o globo. Um exemplo é o projeto Asgard (https://github.com/Netflix/asgard) , desenvolvido pela Netflix que é o responsável por gerenciar todo o processo de deploy de suas apli-

cações na nuvem, além de diversos outros projetos. Outro exemplo legal? O mecanismo de build adotado por padrão para aplicações Android hoje é o Gradle (http://www.gradle.org) , inteiramente implementado em... Groovy. Estes são apenas alguns exemplos, mas não se assuste se um dia descobrir que já usa a linguagem há um bom tempo sem saber, visto que é componente de diversas bibliotecas e frameworks de sucesso entre programadores Java como, por exemplo, o Jasper Reports e o Spring.

Groovy é uma linguagem dinâmica: isto quer dizer que podemos alterar o comportamento do nosso código enquanto este é executado. Soa estranho em um primeiro momento mas, como veremos no decorrer deste livro, trata-se de um aspecto poderosíssimo que nos permite escrever programas que seriam muito mais trabalhosos em Java convencional. Esta é, inclusive, a principal característica da linguagem que propiciou a criação do Grails.

Como veremos no capítulo seguinte, linguagens dinâmicas nos permitem executar tarefas em tempo de execução que em outras linguagens só são possíveis no momento da compilação ou através de padrões de projeto ou soluções arquiteturais que nem sempre são as mais amigáveis.

Adotar o paradigma funcional de programação também é mais fácil graças à presença das tão faladas *closures*, um recurso que só apareceu na versão 8 do Java. Veremos o enorme leque de possibilidades que elas nos propiciam, como a redução de código *boilerplate* que normalmente precisamos escrever em Java.

Também não podemos deixar de citar as melhorias sintáticas oferecidas pelo Groovy: há construtores que nos possibilitam, por exemplo, lidar com as coleções Java (a API *Collections*) de uma forma muito mais simples e eficiente. Além disto, entram como agregados interessantes uma nova sintaxe para lidar com strings, a possibilidade de sobrecarregarmos operadores, lidar com números do tipo `BigDecimal` de forma mais natural, estruturas de controle aprimoradas e muitas outras facilidades que tornam a escrita de código uma tarefa muito mais prazerosa e simples do que a experiência que temos com o Java tradicional. Pense em todas as coisas na linguagem Java de que você não goste e as exclua: este é o Groovy.

Uma crítica frequente à linguagem dizia respeito ao seu desempenho, que realmente era consideravelmente inferior ao do Java. Esta, entretanto, é uma

realidade bastante diferente hoje em dia. Com os avanços em seu desenvolvimento, podemos dizer que na versão 2.4 é muito próxima à do Java, o que a torna uma alternativa viável para a implementação da maior parte dos projetos voltados para a JVM desenvolvidos atualmente. Além disso, minimiza a curva de aprendizado desta por aqueles que já conheçam o Java tradicional.

(Se bem que toda conversa sobre desempenho só faz sentido quando existe um requisito que nos diga qual o desempenho que nosso sistema deverá apresentar.)

Para finalizar, a linguagem Groovy possui outra característica que a torna atraente aos olhos dos desenvolvedores: o fato de ser completamente compatível com seu código Java existente. Com isto, o desenvolvedor pode reaproveitar toda a bagagem existente na plataforma Java, que atualmente possui a maior biblioteca de código-fonte desenvolvida por terceiros na história da computação. Seu código Java pode ser acessado de forma transparente pelo seu código Groovy e vice-versa.

Há, no entanto, um aspecto bastante negativo na linguagem Groovy que já devo adiantar ao leitor. Uma vez adaptado ao seu modo de trabalho, acostumar-se novamente ao modo de programar oferecido pela linguagem Java torna-se uma tarefa bastante penosa. :)

1.3 Por que Grails?

Grails é um framework para desenvolvimento de aplicações web profundamente influenciado pelo Ruby on Rails (seu primeiro nome sequer era Grails, mas sim "Groovy on Rails"). Para entender os ganhos que ele nos trouxe, primeiro é preciso lembrar do mundo em 2004 quando Rails chocou a todos nós.

Ao ser lançado em 2004, Ruby on Rails mudou para sempre a maneira com a qual estávamos acostumados a escrever aplicações web para a plataforma Java (e nem era voltado para Java!). Foi um tapa na cara de todos nós, desenvolvedores Java EE: de repente víamos diante dos nossos olhos como era possível desenvolver aplicações web sem a necessidade de enfrentar incontáveis arquivos de configuração, tal como estávamos acostumados a fazer. Em vez disso, foi apresentado algo comum em outras plataformas: o desen-

volvimento baseado em convenções e em uma linguagem dinâmica, no caso Ruby.

Até aquele momento, estávamos acostumados ao "modo Java" de trabalhar: arquivos de configuração `XML` que atingiam tamanhos difíceis de serem gerenciados, com um código burocrático que precisava implementar uma ou mais interfaces.

Além disso, havia um ambiente de desenvolvimento baseado em um ciclo do tipo escreve, compila, instala no servidor e o reinicializa para ver se funcionou direito o código implementado. Um procedimento tedioso, lento e que minava completamente a produtividade do desenvolvedor.

Grails foi uma das respostas apresentadas a esta situação, cuja versão 1.0 foi lançada em 2008. Era como trazer o Ruby on Rails para a plataforma Java. Assim como naquele ambiente, tínhamos um framework de pilha completa (*full stack*), ou seja, o desenvolvedor não precisava se preocupar com a integração de suas diversas bibliotecas, como o Hibernate para persistência, Log4J para logging, Spring e muitas outras. Tínhamos ali todas as estrelas do Java finalmente integradas, prontas para o uso: o desenvolvedor podia começar a implementar sua lógica de negócio imediatamente.

Tínhamos também um processo de desenvolvimento bem mais ágil: o antigo ciclo escreve-compila-instala-reinicia virou passado. Alterações em nosso código-fonte imediatamente eram refletidas na aplicação em execução, possibilitando ao programador ver de maneira imediata o resultado de suas alterações em tempo de execução.

E os arquivos de configuração eram, em sua maior parte, apenas uma triste lembrança do passado. Como a maior parte das bibliotecas com as quais estávamos habituados a trabalhar já estavam prontamente integradas, tudo o que precisávamos fazer (e mesmo assim opcionalmente) era configurar a nossa conexão com o banco de dados.

- O *scaffolding* que automaticamente gera para o desenvolvedor as camadas de visualização e controle tendo como base nossas classes de domínio.

- Injeção de dependências e inversão de controle baseada em Spring.

- Diversas das principais bibliotecas e frameworks Java prontamente integrados, de tal modo que não precisamos nos preocupar mais com isto.
- Uma arquitetura de plug-ins que nos permite modularizar nossos projetos e enriquecer o framework de uma forma bastante interessante.
- É baseado em uma das linguagens mais produtivas já criadas: Groovy.
- E não podemos esquecer de mencionar nossa vibrante comunidade, tanto nacional quanto internacional.

O modo como lidamos com a persistência e validação de dados também é mais simples com Grails. Temos o GORM, nosso framework para persistência, que atualmente trabalha tanto com bases de dados relacionais como não relacionais, como o MongoDB. Não é necessário ao desenvolvedor se preocupar com anotações ou arquivos de mapeamento. Como veremos, é um procedimento simples e direto. O desenvolvedor só precisa lidar com o modo como suas classes são modeladas, como deveria ser desde o início.

Por trás dos panos temos o Spring, atualmente um dos melhores (talvez o melhor) *container* de inversão de controle e injeção de dependências do mercado. Para a surpresa de muitos, este raríssimas vezes será visto de forma direta pelo desenvolvedor. Seu uso também foi radicalmente melhorado e injetar dependências nunca foi tão fácil e transparente.

É interessante observar que o acesso a diversas funcionalidades do Java EE também foi simplificado com Grails. Para cada recurso da plataforma corporativa Java há uma interface Grails que facilita o seu uso para todos aqueles que possuem interesse em trabalhar com Java EE mas se sentem intimidados por sua aparente complexidade. Grails resolve de uma forma bastante elegante este problema.

O mais importante é que, finalizado seu projeto Grails, o resultado é uma aplicação Java EE completa, com os mesmos direitos que qualquer uma escrita para a plataforma possui: alta disponibilidade, escalabilidade, desempenho e robustez. E, como brinde, há também o fato de o código Grails ser altamente portável entre servidores de aplicação, evitando-lhe este tipo de dor de cabeça no futuro.

Trata-se de um framework direto ao foco: o desenvolvedor vê minimizado o tempo que até então era gasto com complexidades não funcionais, como integração de bibliotecas e gerência de servidores.

A partir de agora, o foco passa a ser no que realmente interessa, o sistema que precisa ser escrito, e não mais em problemas periféricos.

A comunidade Grails tem crescido bastante no Brasil. Quando fundei o Grails Brasil (http://www.grailsbrasil.com.br) em 2008, logo no primeiro mês, para minha surpresa, éramos 60. Em 2015, contamos com mais de 2100 membros (e o número não para de crescer) que participam ativamente ajudando aqueles que estão iniciando, resolvendo suas dúvidas e problemas, escrevendo novos plug-ins e funcionalidades para o framework.

Assim como Groovy torna a escrita de código um processo mais simples e pragmático, o mesmo observamos com Grails, que acaba com o tédio no processo de desenvolvimento de aplicações web para a plataforma Java.

1.4 Sobre este livro

A maior dificuldade que enfrentei ao escrever este livro diz respeito ao seu formato. Não acredito que a natureza dinâmica de Groovy e Grails possa ser passada usando o formato de prosa tradicional, por esta razão optei pelo diálogo como meio principal de narração. Em minha experiência como consultor nestas e outras tecnologias, já constatei que o processo de aprendizado se dá de uma forma muito mais natural e eficiente quando a prosa entra em cena e as formalidades se tornam apenas um acessório (necessário).

Ao lê-lo, pense em um grande diálogo comigo e as personagens que entrarão em sua vida durante a experiência. As conversas que você acompanhará são reais e as experimento sempre que apresento estas tecnologias. Esqueça o formato “manual” padrão: ao falarmos de Groovy e Grails, o que realmente importa é a interação e o modo como lidamos uns com os outros.

Aliás, sabemos que este modo de descrever o mundo funciona muito bem desde a Grécia antiga (lá por volta de uns 300 a.C.), que exerceu fortíssima influência na escrita deste livro. Não há inovações no que fiz aqui: apenas resgatei uma prática antiga.

Algumas partes do livro devem, na minha opinião, ser lidas com aten-

ção redobrada. No caso, refiro-me aos capítulos sobre Groovy e GORM. Na maior parte dos projetos em que vi a aplicação do Grails fracassar, o principal vilão foi sem sombra de dúvidas o não conhecimento destas duas tecnologias. "Pense Groovy" ao usar Grails, e fique atento às "armadilhas" que aponto durante todo este trabalho.

É importante lhe avisar que uso o termo "armadilha" não é para apontar bugs na plataforma, mas sim para salientar aspectos do seu funcionamento que não são conhecidos pela vasta maioria dos programadores e que muitas vezes poderá custar-lhe inúmeras horas que poderiam ser gastas em atividades mais úteis. Sendo assim, ao ver o ícone a seguir, já sabe: atenção redobrada!

Fig. 1.1: Atenção redobrada!

Durante a escrita deste livro usamos a versão 2.4.4 do Grails e o release 3.0.0 estava se aproximando e finalmente foi lançado no dia 31/3/2015. Dado ser uma versão na qual grandes partes do framework foram reescritas, a quantidade de bugs ("oportunidades de melhoria") costuma ser significativa, razão pela qual não lhe recomendo adotar esta versão neste momento (abril de 2015), mas sim em um futuro próximo, quando saírem as versões 3.0.4 ou posteriores. Mas isso não quer dizer que estamos lançando um livro desatualizado, pelo contrário!

Em diversas páginas você encontrará o ícone a seguir:

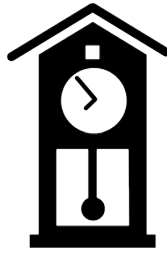



Fig. 1.2: Compatibilidade

Ele indica que irei falar sobre o que será diferente na versão 3.0 do Grails ou do modo como versões antigas funcionavam. Além disto, também me preocupei em incluir como capítulo final uma descrição das principais mudanças trazidas pela versão 3.0. Não se assuste: são poucas e praticamente tudo o que veremos nestas páginas se aplica quase que sem alteração nenhuma na nova grande versão.

1.5 Código-fonte

Neste livro, iremos desenvolver uma aplicação cujo código-fonte pode ser baixado no GitHub neste link: https://github.com/loboweissmann/FalandoDeGrails

Tratam-se de dois projetos: *ConCot*, que é a aplicação principal (você verá mais sobre ela mais à frente) e *DDL-Plugin-Itens*, que é um plug-in que desenvolveremos. O funcionamento do primeiro depende do segundo, sendo assim, basta copiar o diretório raiz do projeto para um local de sua preferência, que tudo deverá funcionar perfeitamente.

Caso tenha algum problema, já sabe: pode entrar em contato diretamente comigo por qualquer uma das fontes a seguir:

- E-mail: kico@itexto.com.br
- Twitter: @loboweissmann

1.6 Agradecimentos

Esta é aquela parte do livro na qual tenho certeza de que serei injusto pois a quantidade de pessoas que me incentivaram e apoiaram na conclusão deste trabalho é **muito** grande.

Para começar, agradeço a todos os amigos que fiz na comunidade Java e Groovy, especialmente no Grails Brasil (http://www.grailsbrasil.com.br) . Quando me lembro de como as coisas se iniciaram em 2008, naquele fórum que criei usando phpBB (sim, PHP), que começou a reunir pessoas que queriam ajudar outros colegas e aprender aquele framework e linguagem que nos pareciam tão alienígenas, e que hoje vejo sendo aplicados em uma imensidão de empresas e projetos, confesso ficar sem fôlego. Agradeço a todos vocês.

Agradeço também à paciência (aparentemente infinita!), suporte e atenção que Adriano Almeida e Paulo Silva, editores da *Casa do Código* que, sem exagero, considero verdadeiros heróis que alavancam a propagação do conhecimento em nossa área, mesmo com todas as dificuldades e que inclusive apostaram no formato diferenciado que escolhi para a escrita deste livro.

Não tem como não agradecer também à melhor coisa que ocorreu em minha vida que se chama Nanna (oficialmente conhecida como Maria Angélica Álvares da Silva e Silva). Tantos cafés, discussões e suporte que não tenho palavras para descrever.

Ah, claro: não tem como não agradecer ao Platão, Trasímaco, Xenofonte e alguns outros que influenciaram de forma profunda o modo como decidi escrever este livro e, claro, *Daniel*, *Guto* e *Dedé* que sem saber acabaram entrando nesta história.

Para evitar que eu sofra "consequências imprevisíveis", novamente agradeço à minha mãe, Selma Weissmann. :)

Obrigado, obrigado e mil vezes obrigado a todos vocês!

E mil desculpas a qualquer um que eu não tenha citado diretamente aqui.

(Se bem que ninguém lê a seção de agradecimentos mesmo.)

Capítulo 2

Groovy: uma breve introdução

Groovy é uma linguagem de programação que tem uma sintaxe bastante parecida com a do Java, o que traz prós e contras: por um lado, facilita sua adoção por estes programadores dada sua similaridade e também por suprir diversas deficiências da linguagem. Por outro, acaba gerando problemas para aqueles que começam a dar os primeiros passos com Grails: não é raro o programador acreditar estar lidando com "uma espécie de Java" e, com isto, perder a chance de tirar máximo proveito da linguagem e, consequentemente, do próprio Grails.

Caso queira adicionar mais uma linguagem à sua caixa de ferramentas (o que sempre recomendo), bem vindo(a)! Torço para que ao final da leitura dos capítulos sobre Groovy você se sinta motivado a adotar esta que, acredito, seja uma das mais produtivas linguagens de programação que conheço.

2.1 Para que outra linguagem de programação?

Uma pergunta corrente que ouvimos sobre Grails é: será que o framework não seria muito mais popular se fosse baseado na linguagem Java em vez de Groovy, dado que o número de programadores Java é **bem** maior? Para responder a esta pergunta, basta olharmos para o ano em que Grails chegou à versão 1.0: 2008. Nessa época a versão mais usada da linguagem Java era a 6.0, lançada dois anos antes. A pergunta que se faz é: o que faltava no Java?

Faltavam closures, facilidades de metaprogramação facilmente alcançáveis com linguagens dinâmicas e diversas outras funcionalidades das quais a implementação do Grails usa e abusa. Todas estas ausências do Java poderiam ser superadas usando a própria linguagem a partir da implementação de novas bibliotecas ou APIs, mas será que o custo na implementação de todo este código valeria a pena? Se valesse, hoje teríamos apenas Java executando na JVM, no entanto não é o que observamos quando olhamos para linguagens como Groovy, Clojure, Scala, JRuby, Jython e tantas outras.

Esta pergunta nos leva a um tema muito interessante da Ciência da Computação: a questão da equivalência e completude de Turing [30] (*Turing Complete* e *Turing Equivalence*). Na década de 1930, o matemático inglês Alan Turing nos trouxe sua famosa *Máquina de Turing*, um modelo matemático que definiu o modo como "tudo o que é computacional" funciona até os dias atuais. Esta é uma teoria tão profunda que deu origem inclusive a um ramo da ciência chamado *Física Digital*, que diz ser todo o Universo descrito por informação e, portanto, computável e passível de ser tratado digitalmente (e nem mencionei a computação atual).

A máquina de Turing nos fornece as operações básicas por trás da computação, e um elemento é dito *Turing complete* quando as consegue executar. Uma linguagem de programação orientada a objetos como Groovy e Java são *Turing complete*. E de onde vem este papo de equivalência? Se a linguagem A é *Turing complete* assim como a B, então os procedimentos de A podem ser convertidos em B e vice-versa. A consequência imediata é que na essência o que você pode fazer com Groovy ou Scala também pode ser feito em Java e vice-versa, ou seja, todas as linguagens de programação que caibam nesta categoria são equivalentes. Então, por que não ficamos apenas no Fortran? Por que criamos tantas linguagens de programação?

A resposta é simples: notação e comodidade. Sim, eu posso obter o mesmo resultado computacional com Java no Fortran, mas o trabalho que eu tenho para tal é enorme e simplesmente não compensa. Esta é a razão pela qual novas linguagens de programação surgem: assim, você pode fazer mais com uma notação mais agradável ou próxima do objetivo a que seu programa visa. É possível termos orientação a objetos em C? Sim. É mais trabalhoso com C ou Java?

Escrever código dinâmico é mais fácil em Groovy ou Java? Como veremos neste capítulo e nos próximos, com certeza em Groovy.

2.2 Afinal de contas, o que é o Groovy?

Após toda esta teoria, a pergunta óbvia ainda não foi respondida: o que é Groovy? O básico o leitor já sabe: trata-se de uma linguagem de programação. Assim como Java, é orientada a objetos e desenvolvida para que seja executada na *máquina virtual Java* (JVM).

A Máquina Virtual Java

O que permite que um programa escrito em Java seja executado em qualquer sistema operacional é a Máquina Virtual Java, também conhecida como JVM (*Java Virtual Machine*). Ela basicamente simula um computador real que, em vez de executar código de máquina nativo como ocorre com seu relativo físico, interpreta *bytecode*.

Bytecode é a língua franca entendida pela JVM. Todo código-fonte Java é convertido para este formato, que é independente de sistema operacional. Praticamente tudo o que um sistema operacional precisa para poder executar código Java é possuir uma implementação da JVM, portanto.

Mas há mais diferenças: Groovy é uma linguagem dinâmica. Isso quer dizer que consegue executar tarefas que em linguagens estáticas só se consegue em tempo de compilação ou adotando técnicas como padrões de projeto que nem sempre são simples. Que tarefas são essas? Basicamente, modificar

código executável em tempo de execução. É possível, por exemplo, incluir novos comportamentos (métodos e funções) em nossas classes ou alterar código preexistente. Mais que isto, no caso do Groovy, ainda temos acesso à API de reflexão de uma forma muito mais simples e direta do que no Java tradicional.

Na semelhança também há diferenças: Groovy nos apresenta uma sintaxe muito próxima do Java, reduzindo a curva de aprendizado, mas ao mesmo tempo a aprimora, simplificando tarefas até então chatas da linguagem. A grosso modo, não é exagero dizer que Groovy é uma espécie de *"Java++*

. Pense em algo que seja verboso em Java, como lidar com a classe `BigDecimal`. Um rápido exemplo: imagine que queiramos implementar o delta de uma equação de segundo grau em Java. Escreveríamos algo como o código a seguir:

```
BigDecimal delta(BigDecimal a, BigDecimal b, BigDecimal c) {
    return b.multiply(b).subtract(new BigDecimal(4).multiply(a).multiply(c
}
```

E em Groovy?

```
BigDecimal delta(BigDecimal a, BigDecimal b, BigDecimal c) {
    (b*b) - (4*a*c)
}
```

Como veremos neste capítulo e no próximo, há diversas outras melhorias apresentadas pela linguagem. A propósito, outra mudança interessante é que apenas se você quiser a tipagem é estática. Soa estranho para quem está acostumado com Java, mas é um estilo de programação bastante produtivo.

Para satisfazer sua curiosidade, nossa versão do delta pode ser escrita em Groovy tal como o código a seguir. Groovy descobrirá em tempo de execução com qual tipo deve trabalhar e, acredite, sempre funciona! :)

```
def delta(a,b,c) {
    (b*b) - (4*a*c)
}
```

É também uma linguagem de script. Ao contrário do Java em que todo código executável obrigatoriamente precisa estar contido em uma classe, o mesmo não ocorre em Groovy. Curioso de novo? Compare os dois códigos a seguir com o batido exemplo do "Olá mundo":

```
public class OlaMundo {
    public static void main(String args[]) {
        System.out.println("Oi Mundo!");
    }
}
```

E em Groovy?

```
println "Oi Mundo!"
```

Ser uma linguagem de script que executa na JVM torna Groovy uma excelente ferramenta para manutenção e administração de sistemas Java. Por quê? Porque código Groovy no final das contas vira bytecode e acessa transparentemente seu código Java legado. Sim: ainda há esta vantagem. **Todo o seu código Java pode ser usado pelo Groovy e vice-versa**. Outra vantagem deste aspecto "script" é que aquele ciclo de desenvolvimento escreve-compila-executa vira simplesmente escreve-executa, o que aumenta bastante a sua produtividade. Veremos mais sobre isto neste capítulo.

Muita teoria foi exposta, é hora de irmos à prática.

2.3 Instalando Groovy

Instalar o Groovy é bastante simples. Vou expor aqui um procedimento rápido que pode ser aplicado a qualquer sistema operacional.

1) Baixe a última versão do Groovy em http://www.groovy-lang.org/download.html. Opte pela opção no formato `zip`;

2) Descompacte o arquivo em um diretório de sua preferência;

3) Crie uma variável de ambiente chamada `GROOVY_HOME` que aponte para o diretório no qual você descompactou sua distribuição do Groovy;

4) Caso exista uma variável de ambiente chamada `JAVA_HOME` em seu computador, certifique-se de que ela aponte para uma instalação do JDK (é requisito para a instalação do Groovy);

5) Adicione o diretório `GROOVY_HOME/bin` ao `PATH` do seu sistema.

Após esses passos, abra sua interface de linha de comando e execute o comando *groovyconsole*. Se tudo der certo, surgirá uma janela similar à que exponho a seguir.

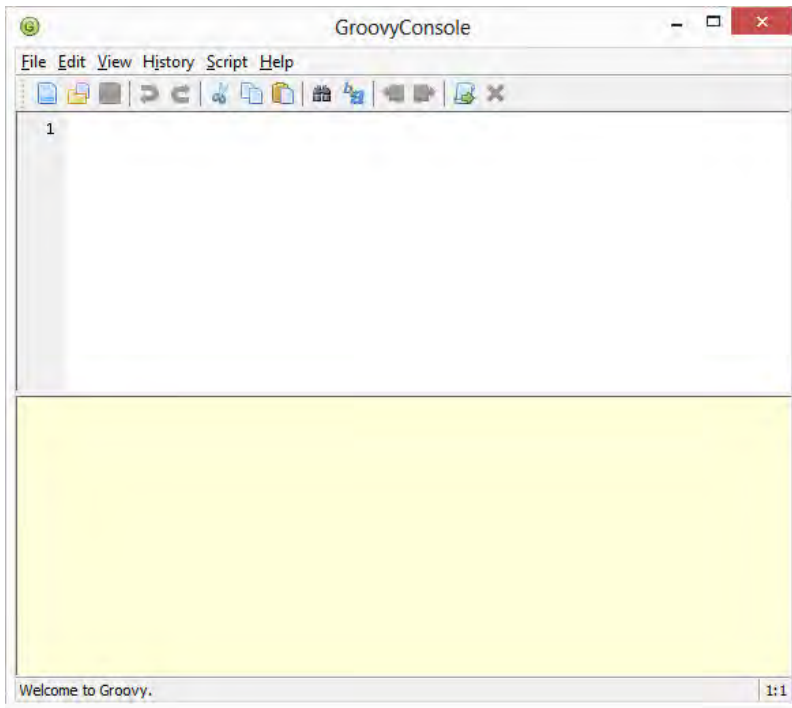



Fig. 2.1: Groovy Console: nosso amigo nos próximos capítulos

2.4 Groovy Console: seu laboratório pessoal

O Groovy Console é uma poderosa ferramenta e nos guiará neste livro daqui para frente enquanto nosso foco for a linguagem Groovy. Este é o que normalmente chamamos de *REPL*. REPL é um acrônimo para *Read Eval Print Loop*, ou seja, uma interface que usamos para executar rapidamente nosso código-fonte e ter acesso imediato ao resultado do seu processamento. A imagem a seguir ilustra muito bem seu funcionamento. Digitamos nosso código na parte de cima da janela; ele é executado e vemos imediatamente o resultado

do programa.

GroovyConsole
File Edit View History Script Help

```
1 def soma(a,b) {
2     a + b
3 }
4
5 soma(3,4)
```

Seu código fonte entra aqui

```
groovy> def soma(a,b) {
groovy>     a + b
groovy> }
groovy> soma(3,4)

Result: 7
```

E o resultado da execução sai aqui

Execution complete. 5:10

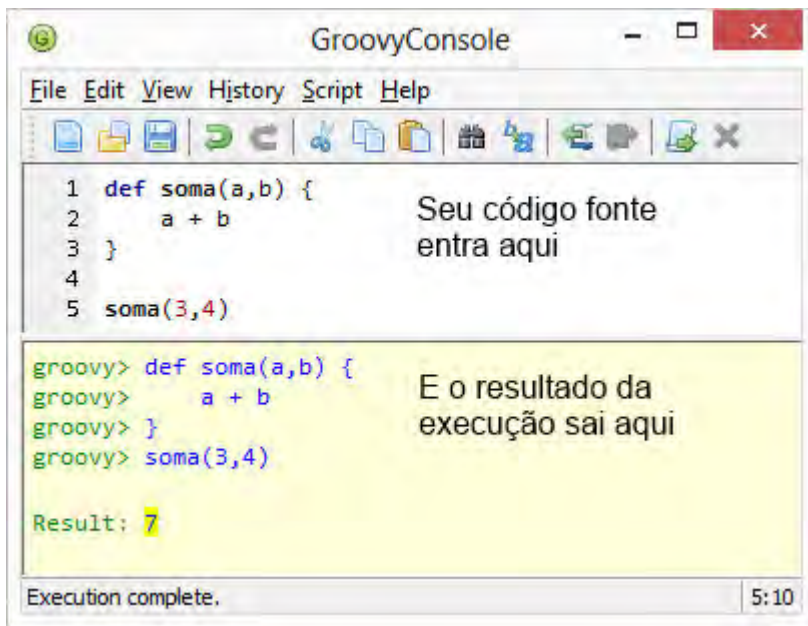



Fig. 2.2: A interface do Groovy Console

Para executar seu código-fonte pelo Groovy Console há dois caminhos. Você pode ir ao menu `Script` e clicar sobre a opção `Run` ou simplesmente usar o atalho de teclado `Ctrl + R`. Outro atalho muito útil é `Ctrl + W`, que limpa a parte de baixo da janela, facilmente poluída conforme vamos usando o REPL em nossos experimentos.

Veja o Groovy Console como seu laboratório pessoal. Ele é uma excelente ferramenta que você poderá usar para esboçar algoritmos e testar ou aprender alguma API que lhe interesse.

Caso esteja trabalhando em um ambiente privado de interface gráfica, há uma alternativa para o Groovy Console: trata-se do comando *groovysh*, que iniciará um REPL no modo somente texto para você, tal como ilustrado na imagem a seguir:

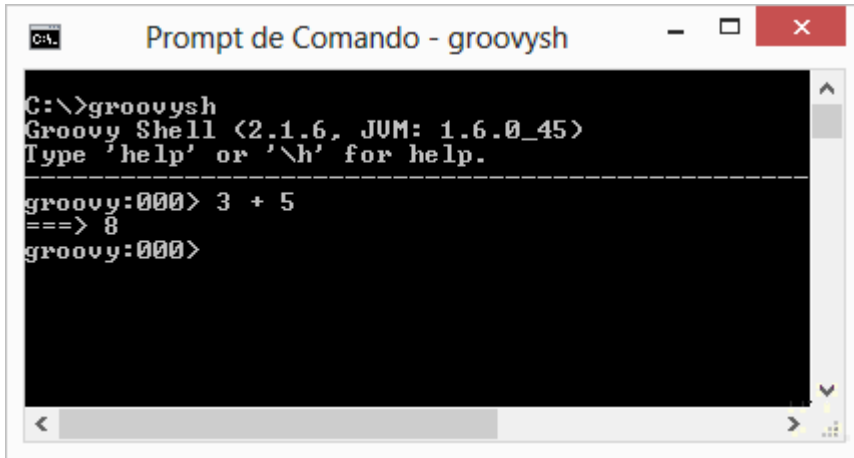



Fig. 2.3: Groovysh: a versão textual do Groovy Console

2.5 "Olá mundo"

Toda introdução a uma linguagem de programação começa com o exemplo mais repetido da história da computação, o "Olá mundo". Na realidade já o expus na seção 2.2, mas é interessante repeti-lo, para que possamos entender o que está acontecendo.

```
println "Olá mundo"
```

Não é preciso envolver nosso código em uma classe como ocorre em Java ou uma função `main` como no C. O método `println` também nos diz algo. Parece-lhe familiar? Talvez você já o conheça do Java, como no exemplo a seguir, que também funciona perfeitamente em Groovy:

```
System.out.println("Olá mundo");
```

O mesmo código poderia também ser escrito da seguinte forma em Groovy:

```
println "Olá mundo"
```

Isto mesmo, parênteses são opcionais em Groovy. O mesmo pode-se falar a respeito do famoso caractere de ponto e vírgula (;), também opcional. Uma versão mais próxima do Java do nosso "Olá mundo" pode ser vista a seguir e é igualmente válida:

```
println("Olá mundo");
```

Então, quando ponto e vírgula é usado? Quando você quiser incluir dois ou mais comandos em uma mesma linha:

```
println "Oi mundo"; println "Tudo bem com você?"
```

A propósito, o último ponto e vírgula também é opcional neste caso, já que não teremos mais nenhuma instrução nessa linha.

2.6 Variáveis: tipagem "opcional"

Variáveis são espaços de memória para armazenar valores usados por nossos programas. Toda variável possui três atributos: tipo, nome e valor. Quando falamos em Groovy, o primeiro destes atributos o tipo pode ser descoberto em tempo de execução. Podemos até mesmo não o declarar, mas por baixo dos panos, não se iluda, há um tipo.

O tipo de uma variável pode ser algo como textos, números, booleanos e muitos outros. A declaração de uma variável em Groovy pode ser feita tal como estamos acostumados em Java:

```
int numero
```

No entanto, visto que o tipo é descoberto em tempo de execução quando estamos falando de Groovy, este pode ser omitido na declaração de uma variável. Para tal, usamos a palavra reservada `def`, que na prática significaria algo como *não me importa saber qual o tipo desta variável, este será definido em tempo de execução*. O código a seguir expõe bem este comportamento:

```
def variavel

// variável passa a armazenar um valor numérico
variavel = 4
```

```
// variável passa a armazenar um valor textual
variavel = "Algum texto"
```

Caso o valor da variável seja definido em sua declaração, a palavra reservada `def` se torna opcional. O código a seguir é perfeitamente válido:

```
pi = 3.14 // automaticamente vira BigDecimal
```

As mesmas regras aplicadas na declaração de variáveis valem também na definição de parâmetros em uma função. Podemos omitir tanto o tipo de retorno quanto o tipo de cada um dos parâmetros:

```
def soma(a,b) {
    return a + b
}
```

Esses são apenas alguns simples casos onde o código Groovy é bem mais enxuto do que o escrito em Java.

2.7 Trabalhe com Strings

Usamos o tipo de dados `String` quando lidamos com informação textual e este é um aspecto do Groovy que costuma cativar diversos programadores Java por tornar a sua manipulação muito mais simples. Aliás, não mais simples, mas sim como sempre deveria ter sido.

A forma mais tradicional de declarar uma `String` em Groovy é exatamente como faríamos em Java.

```
String linguagem = "Groovy"
```

Não se engane, não estamos lidando aqui com a mesma string que usamos em Java, mas sim uma `GString` (`Groovy String` `groovy.lang.GString`), que traz consigo alguns truques bem interessantes, a começar pela concatenação, que agora fica bem mais simples. Em Java, concatenamos assim:

```
String prefixo = "Programando em ";
String sufixo = "Groovy";
String fraseCompleta = prefixo + sufixo;
// Resultado: "Programando em Groovy"
```

Em Groovy, podemos fazer exatamente como em Java, mas por que concatenar se podemos *interpolar*? Assim como em PHP e Ruby, Groovy nos permite definir pontos em uma `String` que serão preenchidos com o que quisermos em seu interior. Confuso? Veja o exemplo a seguir:

```
String sufixo = "Groovy"
String fraseCompleta = "Programando em ${sufixo}"

println fraseCompleta
// resultado: "Programando em Groovy"
```

No interior de `${}`, deve ser incluída uma expressão qualquer, que pode ser tanto o valor de uma variável como o resultado de uma expressão matemática ou lógica:

```
int a = 3
int b = 5
String expressao = "${a} + ${b} = ${a + b}"
println expressao
// resultado: "3 + 5 = 8"
```

Talvez você não queira executar uma expressão tão simples como retornar o valor de uma variável. Neste caso, tudo o que você precisa fazer é usar a sintaxe `$`, que é bastante simples:

```
int a = 3
def texto = "Qual o valor de a? É $a"
println texto
//resulta em
//"Qual o valor de a? É 3"
```

Basta que seguindo o caractere `$` seja incluído, sem espaços, o nome da variável cujo valor queremos interpolar em nossa `String`. Simples assim.

Há outra tarefa que faz parte do cotidiano de todo programador e que normalmente é bastante chata: lidar com strings que sejam compostas por mais de uma linha. Em Java, não é raro encontrar código como este:

```
String muitasLinhas = "Olá mundo!\n" +
                      "Estou aprendendo Groovy.\n" +
                      "E me encantando com a linguagem. :)";
```

Em Groovy, basta declararmos nossa `String` não com aspas duplas, mas sim triplas como no exemplo a seguir:

```
String muitasLinhas = """Olá mundo!
Estou aprendendo Groovy.
E me encantando com a linguagem. :)"""
```

E, sim, também podemos usar interpolação:

```
int a = 3
int b = 5
String muitasLinhasInterpoladas = """a = ${a}
b = $b
a + b = ${a + b}"""
```

Ainda há mais um tipo de `String` que podemos usar, a nativa do próprio Java, `java.lang.String`. Claro que há uma sintaxe especial para isto também, as de aspas simples:

```
String stringJava = 'Sou a string padrão do Java'
```

Se a `GString` é tão mais poderosa que a `String` padrão do Java, por que usar a menos poderosa? Performance. Se você não for trabalhar com texto presente em múltiplas linhas ou interpolação, é muito mais vantajoso usar a versão mais simples que supre bem as necessidades do seu código, não é mesmo?

2.8 O que é verdade em Groovy?

O que é verdadeiro em Groovy? Também temos o tipo `boolean`, que representa os valores verdadeiro ou falso. Quando pensamos em Java, temos como verdadeiras as seguintes condições:

- O valor de algo `boolean`;
- O resultado de uma expressão, como comparações.

Em Groovy, o mesmo conceito ainda se aplica, porém expandido, e este é um ponto no qual diversos programadores Java acabam cometendo erros ao se aventurar na linguagem. O objetivo dos criadores da linguagem foi aproximá-la da linguagem coloquial. Para facilitar a exposição deste conceito, primeiro vou lhe introduzir a função `assert` do Groovy. Ela recebe como parâmetro uma expressão e, se esta for verdadeira, nada ocorre, mas caso contrário uma exceção será disparada, tal como no exemplo a seguir:

```
def a = 3
assert a < 4 // nada ocorre: a realmente é menor que 4
assert a != null // nada ocorre
assert a > 5 // uma exceção será disparada
```

A exceção disparada é bastante instrutiva, apontando onde a falha ocorreu, como pode ser visto na imagem a seguir.

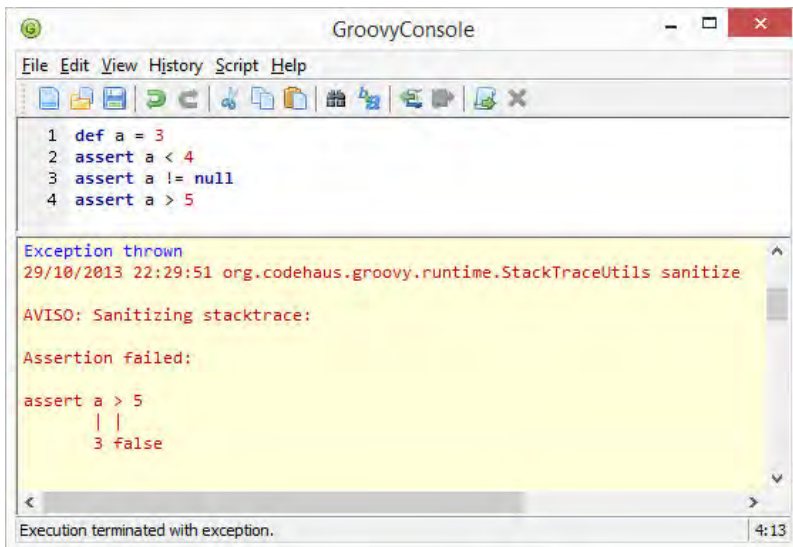



Fig. 2.4: Assert expondo graficamente o erro.

Agora voltemos ao nosso conceito de verdade. Se o valor for nulo, é considerado falso, caso contrário é verdadeiro em Groovy. Isto evita termos de escrever código verboso, o que costumávamos fazer em Java:

```
if (pessoa != null &&
    pessoa.getNome() != null &&
    pessoa.getNome().equals("Henrique")) {
    // faça algo após tudo isto
}
```

Em Groovy a coisa fica bem mais simples:

```
if (pessoa && pessoa.nome) {
    // faça algo
}
```

Mas sabia que dá pra ficar ainda **mais simples**?

```
if (pessoa?.getNome()) {
    //hein?
}
```

O caractere `?` é o que chamamos em Groovy de operador de navegação segura (*safe navigation operator*). O que isso faz é evitar a famosa `NullPointerException`, que ocorre quando tentamos acessar um atributo ou função de um objeto que seja nulo. O código a seguir expõe bem esta facilidade:

```
def pessoa = null
pessoa?.grite() // NullPointerException não será disparada :)
```

Já sabemos que expressões booleanas e nulidade de objetos nos dizem se uma expressão é verdadeira ou falsa. Mas o mesmo também se aplica a `Strings`. Uma `String` vazia é considerada falsa:

```
def texto = "texto"
assert texto // teste passa: verdadeiro, texto não vazio
def textoVazio = ""
assert textoVazio // teste não passa: texto vazio
```

O conceito de verdade também se aplica a valores numéricos. Assim como na linguagem C, se um número for igual a zero, ele será falso. Caso contrário, será considerado verdade:

```
assert  1 // teste passa: verdadeiro pois é diferente de zero
assert -1 // teste passa: verdadeiro pois é diferente de zero
assert 0  // teste falha
```

A coisa fica interessante quando precisamos lidar com coleções (veremos coisas interessantíssimas que Groovy nos possibilita fazer com esta API). Se uma coleção for vazia, então a expressão é considerada:

```
def lista = new ArrayList()
lista.add("Valor")
assert lista // teste passa: lista não vazia

lista.clear() // remove todos os itens

assert lista // teste falha
```

Caso um iterador não possua mais itens, sua avaliação também retornará falso:

```
def lista = new ArrayList()
def iterator = lista.iterator()
lista.add("valor")

assert iterator // verdadeiro: iterator tem itens

iterator.next() // fui para o final da lista

assert iterator // falso: sem mais itens
```

2.9 Estruturas de controle

É difícil pensar em um programa útil que não possua uma estrutura de controle. No caso do Groovy temos as mesmas estruturas com as quais estamos acostumados no Java.

If-then-else

Muito provavelmente, a estrutura de controle mais conhecida seja o `if`. Ela é usada quando precisamos lidar com situações do tipo *"se ocorrer esta condição, faça isto, caso contrário, aquilo*

. No caso do Groovy, essa estrutura funciona exatamente como estamos acostumados em Java:

```
def a = 3
if (a <= 3) {
    println "Sou menor ou igual a 3"
}
// Saída: "Sou menor ou igual a 3"

if (a == 5) {
    println "Sou igual a 5"
} else {
    println "Sou igual a $a"
}
// Saída: "Sou igual a 3"

if (a == 5) {
    println "Sou igual a 5"
} else if (a == 4) {
    println "Sou igual a 4"
} else {
    println "Sou diferente de 5 e 4. Sou $a"
}
// Saída: "Sou diferente de 5 e 4. Sou 3"
```

Outra face da estrutura de seleção `if` é o operador ternário, cuja sintaxe é bastante simples:

```
(condição) ? (ação se verdadeiro) : (ação se falso)
```

O uso mais comum do operador ternário é na atribuição de valores a uma variável, tal como no exemplo a seguir:

```
def nome = "Henrique"
def mensagem = nome.equals("Henrique") ? "Belo nome" : "Outro nome"
assert mensagem == "Belo nome" // teste passa
```

O operador `==` em Groovy equivale a uma chamada à função *equals* da classe `Object` do Java. Reparou como fica mais simples assim?

Apenas para clarificar melhor, outro exemplo do uso do operador ternário, desta vez executando procedimentos.

```
def condicao = 1
condicao != 1 ? condicaoDiferenteDeUm() : condicaoUm()
// executa condicaoDiferenteDeUm
```

Switch: o if escrito de forma diferente

Caso não conheça esta estrutura de controle, pense nela como se fosse uma espécie de *"salvadora de ifs complicados*

. É usada quando precisamos lidar com situações nas quais, de acordo com o valor de uma variável, algo precisa ser feito.

A sintaxe desta estrutura é simples:

```
switch (variavel) {
        case [valor_1]:
            //faça isto
        break
        case [valor_2]:
            // faça aquilo
        break
        default:
            // caso seja qualquer outra coisa, execute isto
}
```

Nossa sintaxe é composta por dois componentes: a instrução `switch` em si, que simplesmente nos diz algo mais ou menos como *"se o valor da variável for...*

seguida de um ou mais casos, representados por trechos de código encapsulados pela instrução `case`. Cada `case` possui dois componentes, que são a instrução, cujo valor corresponderá à condição com a qual devemos comparar o valor, seguida do bloco de código, delimitado pela instrução `break`.

Em nossa sintaxe também há um caso especial chamado `default`. É a condição a ser disparada caso nenhuma das definidas anteriormente seja

satisfeita (repare que a palavra `case` não é usada nesta situação). Lembre-se de quando estiver programando em Groovy sempre incluí-lo como o último caso.

Para entendermos melhor o poder do `switch` vamos para um exemplo prático:

```
def valor = 4 // um tipo numérico
switch (valor) {
    case 5:
        println "Eu sou um cinco."
        break
    case 4:
        println "Sou um quatro!"
        break
    default
        println "Sou alguma outra coisa: $valor"
}
// Saída:
// "Sou um quatro!"
```

No exemplo, podemos ver três comparações: a primeira com o valor numérico 5 e a segunda com o valor 4. Repare no caso default ao final, que seria executado caso o valor de nossa variável fosse qualquer coisa diferente de 4 e 5.

Há algumas surpresas interessantes em uma instrução `switch`. Observe o próximo exemplo:

```
def valor = 4
switch (valor) {
    case "4":
        println "Sim, sou um quatro-string"
        break
    case 5:
        println "Quero tanto ser um cinco!"
        break
    case 4:
        println "Sou o quatro!"
        break
```

```
    default:
        println "Sou qualquer outra coisa"
}
```

O que será impresso? *Sim, sou um quatro-string*. Isso porque, durante a comparação, ao chegar nesta opção, Groovy transformou nosso valor em uma string e, em seguida, o comparou com o `case`. Em uma instrução `switch` lembre-se sempre da seguinte regra: **caso seja encontrada uma condição verdadeira, havendo uma instrução `break` o controle sai do `switch`**.

Há casos, no entanto, em que queremos que nosso `switch` se comporte como um *ou*. Tudo o que devemos fazer é digitar as instruções `case` em sequência:

```
def valor = 4
switch (valor) {
    case "4":
    case 4:
        println "Sou um quatro"
        break;
    default:
        println "Não sou um quatro :("
}
// Saída:
// "Sou um quatro"
```

Há ainda um ponto duvidoso aqui: de fato, como em nossas instruções `switch` pudemos comparar um valor do tipo numérico com uma string e mesmo assim obter um valor verdadeiro? Por trás das cortinas, Groovy adiciona um novo método a diversas classes da API Java, chamado `isCase` que recebe como parâmetro um objeto qualquer. No caso de uma `String`, este é implementado de forma muito similar à exposta a seguir:

```
def isCase(object) {
    return object.toString().equals(this)
}
```

Sendo assim, qualquer classe que implemente o método `isCase` está apta a ser incluída em uma instrução `switch`. Pequenos segredos do Groovy dominados por uma minoria. ;)

Aviso aos Javeiros

A estrutura `switch` em Groovy é bem mais flexível do que a que temos em Java. Enquanto no Java só podemos aplicar o switch em valores do tipo `enum`, inteiro (`Integer`), caractere (`char`) e texto (`String`), em Groovy podemos usar qualquer objeto.

O bloco `default` também é ligeiramente diferente: enquanto em Java este pode ser digitado em qualquer posição dentro da instrução `switch`, em Groovy deve vir sempre no final. A razão pela qual isso ocorre é que, ao contrário do Java, cada bloco `case` possui seu próprio escopo e pode ser aplicado em cima de qualquer tipo de dados.

Loops com while

Toda (ou quase toda) linguagem de programação que se preze deve possuir ao menos uma estrutura de controle que permita a escrita de loops, que são estruturas de controle que garantem que determinado bloco de código seja executado enquanto uma condição for satisfeita.

A forma mais simples de loop em Groovy é a construção `while` cuja sintaxe é:

```
while (condição) {
    // bloco de código a ser executado
}
```

Um exemplo rápido de loop é exposto a seguir:

```
def valor = 0
while (valor < 10) {
    println valor
    valor = valor + 1
}
// Saída:
// 0
// 1
// ...
// 9
```

Loops com for

Uma estrutura mais complexa para construirmos loops é o `for`, que possui duas sintaxes, a primeira exposta a seguir:

```
for (valor in iterável) {
    // faça algo
}
```

Um iterável é qualquer objeto que implemente a interface `java.lang.Iterable`, como listas, conjuntos e diversas outras estruturas. A seguir podemos ver um exemplo desta estrutura:

```
def lista = [1,2,3,4]
for (valor in lista) {
    println valor
}
// Saída:
// 1
// 2
// 3
// 4
```

Há uma segunda forma para esta estrutura, que é bastante parecida com a que estamos acostumados em linguagens como Java ou C. Um pouco mais complexa, porém igualmente útil:

```
for (declare uma variável; condição de parada; altere o valor) {
    //código
}
```

É uma apresentação bem esquisita, mas fica mais fácil de ser compreendida com um exemplo simples.

```
for (i = 0; i < 10; i+=4) {
    println i
}
// Saída
// 0
// 4
// 8
```

Há três passos na execução do loop: primeiro criamos uma variável chamada `i`. Logo em seguida, vem a comparação com uma condição. Se for verdadeira, o bloco é executado. Depois vem a alteração do valor, que é a terceira operação no cabeçalho do `for`.

2.10 Funções e métodos

Se o leitor só pudesse programar com o que foi apresentado até este momento no livro, com certeza todos os seus programas seriam verdadeiros pesadelos de manutenção. A primeira estrutura em uma linguagem de programação que permite o mínimo de modularidade e reaproveitamento de código é a rotina. Há dois tipos de rotinas: método e função. A primeira não retorna um valor enquanto a segunda sim. Pragmaticamente não há grande diferença entre elas, razão pela qual até o final deste livro usarei a palavra *função* para me referir às duas categorias.

Funções são a estrutura que nos permitem um mínimo reaproveitamento de código. Imagine que você precise implementar um cálculo complicado, por exemplo o delta da equação de Bhaskara (b * b - (4* a * c)). Por que ficar repetindo o código o tempo inteiro se podemos organizá-lo em uma função?

```
def delta(a,b,c) {
    return (b*b) - (4*a*c)
}
```

A sintaxe de declaração de uma função é exposta de forma bem esquematizada:

```
[tipo] [nome da função] ([parâmetros opcionais]) {
    //código
    // valor de retorno
}
```

Se a função não possuir parâmetros, basta que sejam incluídos apenas os parênteses após o seu nome. Veja o exemplo a seguir que nos retorna o tempo corrente do sistema em milissegundos.

```
long tempoCorrente() {
    return System.currentTimeMillis()
}
```

Outro ponto importante: a instrução `return` é usada para sair do bloco de código que compõe a função retornando um valor, no entanto é uma instrução também opcional. O retorno de uma função pode ser o valor resultante da execução da última linha do bloco de código.

Poderíamos então reescrever nossa função que retorna o tempo assim:

```
long tempoCorrente() {
    System.currentTimeMillis() // return pra quê?
}
```

Há casos nos quais não queremos ou sabemos qual será o tipo retornado pela nossa função: afinal de contas estamos lidando com uma linguagem que por padrão possui tipagem implícita. Neste caso, basta que o tipo da função seja representado pela palavra reservada `def`:

```
def tempoCorrente() {
    System.currentTimeMillis()
}
```

2.11 Concluindo

Neste capítulo, vimos as estruturas mais básicas por trás do Groovy, porém apenas tocamos o mar com a ponta dos pés. Neste momento você apenas sabe como programar proceduralmente com a linguagem, mas Groovy é uma linguagem orientada a objetos, certo? E, mais que uma linguagem orientada a objetos, é puramente orientada a objetos e também dinâmica. Este é o assunto do próximo capítulo.

Capítulo 3

Mergulhando em Groovy

Até este momento só vimos o essencial do Groovy: nada que justificasse a adoção da linguagem se comparada ao Java ou que nos ajudasse a entender o funcionamento de diversos comportamentos do Grails que veremos no restante deste livro. Neste capítulo vamos mergulhar em Groovy mostrando algumas de suas características e comportamentos que, uma vez conhecidos, lhe tornarão um desenvolvedor Grails mais eficiente e, quem sabe, até mesmo lhe fornecerão ideias para que se torne um programador **Java** melhor também. :)

3.1 Orientação a objetos

Neste livro não iremos ensinar ao leitor os conceitos fundamentais por trás da orientação a objetos, mas sim o modo como Groovy implementa este paradigma e, principalmente, apresentaremos também as principais diferenças

em relação ao Java.

POGOs Plain Old Groovy Objects

Em Java, temos o conceito de POJO (*Plain Old Java Object*), que são classes Java que essencialmente implementam o padrão JavaBeans. No caso do Groovy, temos POGOs (*Plain Old Groovy Objects*) e já aviso ao leitor: muito provavelmente, após tomar conhecimento a seu respeito, programar em Java se tornará uma experiência bastante frustrante. Vamos lá.

O que são JavaBeans?

JavaBeans surgiram como uma convenção criada pela Sun Microsystems para facilitar a criação de componentes reutilizáveis usando a linguagem Java. Não se trata de uma biblioteca ou API, apenas um conjunto de regras que devem ser aplicadas às nossas classes Java para que se adequem ao padrão. As regras são simples:

- Toda classe deve possuir um construtor público.
- Todas as propriedades de uma classe são acessíveis através de métodos `get`, `set` e `is`, o terceiro aplicável apenas a atributos do tipo `boolean`.
- Deve ser implementada a interface `java.io.Serializable`.

Para melhor entender este conceito, vamos primeiro escrever um POJO simples em Java, tal como exposto a seguir:

```
class Pessoa implements java.io.Serializable {
    private String nome;
    private String sobrenome;
    private boolean ativa;

    public String getNome() {
        return this.nome;
```

```
    }

    public void setNome(String valor) {
        return this.nome = valor;
    }

    public String getSobrenome() {
        return this.sobrenome;
    }

    public void setSobrenome(String valor) {
        this.sobrenome = valor;
    }
    // Usamos "is" apenas para atributos booleanos
    public boolean isAtiva() {
        return this.ativa;
    }

    public void setAtiva(boolean valor) {
        this.ativa = valor;
    }
}
```

Agora vejamos como fica a versão Groovy:

```
class Pessoa implements Serializable {
    String nome
    String sobrenome
    boolean ativa
}
```

Em Groovy, todo atributo definido na classe por padrão possui visibilidade pública, mas não se preocupe, pois apesar de parecer, não iremos acessar os atributos diretamente. Observe o script a seguir usando a mesma definição exposta:

```
Pessoa pessoa = new Pessoa()
pessoa.setNome("Henrique")
println pessoa.getNome()
// imprimirá "Henrique"
```

Groovy gera automaticamente para nós todos os `getters` e `setters` para o desenvolvedor. Nossa classe implementou o padrão JavaBeans de forma completamente transparente e com **80% menos código**. Se quisermos, também podemos sobrescrever um método `getter` ou `setter` tal como no exemplo a seguir:

```
class Pessoa {
    String nome
    String sobrenome
    boolean ativa

    String getNome() {
        "meu nome é $nome"
    }
}

Pessoa pessoa = new Pessoa()
pessoa.setNome("Henrique")
println pessoa.getNome()
// imprimirá "meu nome é Henrique"
```

Ainda mais interessante, também não precisamos chamar o método `getter`. Podemos simplesmente referenciar em nosso código o nome do atributo, gerando código muito mais limpo e fácil de ler:

```
Pessoa pessoa = new Pessoa()
pessoa.nome = "Henrique"
println pessoa.nome
// imprimirá "Henrique" (voltamos para a versão anterior da classe)
```

Outro aspecto interessante envolvendo POGOs diz respeito aos construtores. Se o leitor já possui experiência com Java, já deve imaginar que o construtor padrão (sem parâmetros) está implícito na classe, certo? Na realidade, mais do que isto, possuímos construtores dinâmicos em Groovy. Nada melhor do que vê-los em prática:

```
Pessoa pessoa = new Pessoa(nome: "Henrique", sobrenome:"Lobo")
println "${pessoa.nome} ${pessoa.sobrenome}"
// Imprimirá "Henrique Lobo"
```

```
pessoa = new Pessoa(nome: "Henrique", ativa:true, sobrenome:"Lobo")
println "${pessoa.nome} ${pessoa.sobrenome} ${pessoa.ativa}"
// Imprimirá "Henrique Lobo true"
```

Por trás dos panos, quando declaramos os parâmetros do nosso construtor como acabamos de fazer, Groovy está na realidade interpretando-os como se fossem um mapa. O código a seguir possui o mesmo resultado:

```
Pessoa pessoa = new Pessoa([nome:"Henrique", sobrenome:"Lobo"])
```

É apenas um pouco de açúcar sintático que nos ajuda a escrever construtores mais limpos. Em Java, teríamos de escrever inúmeros construtores, um para cada variação que quiséssemos. Em Groovy, basta passar os nomes dos atributos, que a linguagem se encarrega do resto.

Talvez o leitor deseje acessar diretamente os atributos presentes em nossa classe. Isso pode ser feito usando o atributo `@` tal como no exemplo a seguir:

```
class Pessoa {
    String nome
    String getNome() {
        "Meu nome é ${this.nome}"
    }
}
Pessoa pessoa = new Pessoa(nome:"Henrique")
println pessoa.nome
// Imprimirá "Meu nome é Henrique"
println pessoa.@nome
// Imprimirá apenas "Henrique"
```

GroovyBeans ou POGOs?

Em diversas fontes, como o excelente livro *Groovy in Action*, o leitor encontrará referências a GroovyBeans.

GroovyBeans e POGOs na prática são a mesma coisa.

Menos burocracia e mais produtividade

Em Java, quando declaramos uma classe pública, esta deve ter o mesmo nome que o arquivo na qual a escrevemos. Em Groovy, isso não é necessário. Aliás, toda classe declarada em Groovy por padrão é pública e, ainda mais interessante, se quiser, você pode declarar quantas classes quiser dentro de um mesmo arquivo.

O único requisito é que o arquivo possua a extensão `.groovy` em seu nome.

3.2 Traits

Imagine que você tenha desenvolvido um framework para integração de plataformas que tenha se tornado um sucesso. Ao ser lançado, inúmeros desenvolvedores o adotaram como a solução ideal para seus problemas e o usam desde então em todos os seus projetos que requeiram integração.

Neste framework criado por você, há uma interface que deve ser implementada chamada `Integrador`, cuja listagem podemos ver a seguir:

```
interface Integrador {
    boolean iniciarIntegracao()
    boolean finalizarIntegracao()
}
```

Neste momento você está trabalhando na versão 2.0 do seu framework. Idealmente, você quer que todos os seus usuários possam fazer o upgrade da forma mais simples possível: simplesmente substituindo os binários da versão 1.0 pelos da 2.0, mas há um pequeno problema. Sua interface `Integrador` foi ligeiramente modificada. Veja listagem a seguir:

```
interface Integrador {
    boolean iniciarIntegracao()
    boolean finalizarIntegracao()

    boolean testarIntegracao()
}
```

Agora o upgrade se tornou um pouco mais trabalhoso, dado que seus usuários precisarão escrever o método `testarIntegracao` em todas as suas classes que implementam a interface. Como resolver este problema? Com *traits*.

As traits foram incluídas na versão 2.3.0 do Groovy. Podemos defini-las de uma forma bastante simples como sendo "interfaces com métodos padrão", tal como existe no Java 8 (*default methods*) [22]. Mais do que métodos padrão, traits também podem ter atributos (privados ou públicos) e métodos privados. Se seu framework de integração tivesse sido implementado com traits, nossa classe `Integrador` seria similar à exposta a seguir:

```
trait Integrador {
    boolean iniciarIntegracao()
    boolean finalizarIntegracao()
}
```

Na versão 2.0, bastaria que você incluísse um método padrão (*default*) na definição da interface, como no seguinte exemplo:

```
trait Integrador {
    boolean iniciarIntegracao()
    boolean finalizarIntegracao()

    /*
        Nossa integração por padrão é
        válida se foi iniciada e finalizada
        corretamente.
    */
    boolean testarIntegracao() {
        iniciarIntegracao() && finalizarIntegracao()
    }
```

Pronto: agora todas as classes que implementavam a interface `Integrador` na versão 2.0 do framework "ganharão" a implementação do método `testarIntegracao`. Claro, caso seja de interesse de algum usuário do framework, ele poderá sobrescrever este método por outro que lhe seja conveniente:

```
class IntegradorEquipamentos implements Integrador {
    // Ignorado restante da classe
    // para facilitar a leitura

    // Para este caso apenas finalizar
    // já é suficiente
    boolean testarIntegracao() {
        finalizarIntegracao()
    }
}
```

Como o leitor deve ter observado, usamos uma `trait` exatamente como faríamos com uma `interface`, usando a palavra-chave `implements`. Uma trait também pode incluir métodos abstratos, forçando as classes que a implementem a sobrescrever o método:

```
trait TesteEquipamentos {
    // Basta usar a palavra chave abstract
    abstract boolean isLigado()
}
```

Como dito no início da seção, uma trait também pode ter métodos ou atributos privados que não estarão disponíveis para as classes que a implementem. Veja o exemplo a seguir:

```
trait  Comunicador {
    private boolean iniciada

    private void iniciar() {
        if (! iniciada) {
            // executa lógica
            iniciada = true
        }
    }

    void enviarMensagem(String msg) {
        iniciar()
        // envia mensagem
    }
```

```
}

class Telefone implements Comunicador {
    void teste() {
        /*
            Um erro será disparado pois
            o método iniciar só existe
            privado na trait.
        */
        iniciar()
    }
}
```

3.3 Tipagem dinâmica ou estática? Ambas.

Um aspecto interessante do Groovy e que muitas vezes assusta os principiantes na linguagem é o fato de se tratar de uma linguagem cuja tipagem é dinâmica. Esta é apenas a metade da história: a tipagem dos dados pode ser tanto dinâmica ou estática, de acordo com a sua necessidade. Mais do que isto: Groovy permite ao programador habituado a programar em linguagens estáticas como Java a oportunidade de experimentar o outro lado da moeda.

O que diferencia tipagem estática da dinâmica?

Antes de mais nada, o que é tipagem (*type system*)? Como o próprio nome diz, trata-se da caracterização de algo. No caso da ciência da computação, ao usarmos termos como tipos, tipagem, sistemas de tipos etc., nos referimos ao modo como determinada *espécie* de dado é representada em um computador, tanto em alto quanto baixo nível.

Ao declararmos uma variável e atribuirmos a esta um valor tal como no código Java a seguir, estamos informando o computador que ela irá representar um número inteiro, cujo valor inicial é 1979. Estamos também definindo o modo como desejamos que a memória ocupada por esta variável seja manipulada: 32 bits, um dos quais usado para definir o sinal do valor numérico.

```
int valor = 1979;
```

Toda linguagem de programação já vem com um conjunto de tipos de

variáveis predefinidos. O exemplo apenas definiu como desejamos que nossa linguagem de programação interprete o espaço de memória reservado para a variável que acabamos de declarar. Quando definimos *explicitamente* o tipo de uma variável, estamos aplicando o conceito de tipagem estática.

No caso do Groovy, tal como dito no início desta seção, podemos também tirar proveito da tipagem dinâmica:

```
def numero = 1979
```

A palavra-chave `def` usada na definição de variáveis instrui o compilador a interpretar aquela definição como algo similar a *"apenas quero uma variável que se chame 'numero'. Esta deverá conter o valor numérico 1979. Não me interessa se é int ou long, apenas aplique o tipo mais conveniente para a execução do meu código.*

.

A tipagem dinâmica assume a responsabilidade pela escolha correta do tipo de nossas variáveis em vez de contar com a ajuda do programador. Sei que para desenvolvedores habituados com linguagens estaticamente tipadas esta ideia soa assustadora, mas acredite: já faz mais de quarenta anos que conseguimos escrever compiladores capazes de acertar o tipo das nossas variáveis com extrema precisão.

Indo um pouco além, vejamos o exemplo a seguir:

```
def variavel = 1979
println variavel // imprimirá 1979, o número inteiro
variavel = "Um texto qualquer"
println variavel // imprimirá "Um texto qualquer"
```

Pela tipagem dinâmica, a natureza de nossas variáveis será alterada no decorrer da execução do programa. Em um primeiro momento, vimos que tínhamos um número (1979). Logo em seguida, atribuímos à variável uma string como valor e o código funcionou perfeitamente. Esta capacidade de mudança de tipo é o que justifica a inclusão da palavra "dinâmica" em tipagem dinâmica.

Para podermos fixar melhor os conceitos de tipagem estática e dinâmica, duas definições rápidas:

- **Tipagem estática**: o tipo da variável é definido explicitamente antes que esta seja usada.
- **Tipagem dinâmica**: o tipo da variável é definido em tempo de execução.

Tipagem dinâmica e estática: quais as vantagens e desvantagens?

As vantagens da tipagem estática são mais fáceis de se compreender, por isso iniciaremos por elas. A principal diz respeito ao tempo de execução dos nossos programas. Como o compilador já sabe de antemão qual o tipo aplicado a cada valor, ele não precisa se preocupar em descobrir o que é cada variável a cada execução e, ainda mais interessante, possibilita que seja executada uma série de otimizações no código-fonte durante o processo de compilação.

Outra vantagem interessante da tipagem estática está no fato de que com ela podemos detectar em tempo de compilação uma boa gama de problemas que poderiam ocorrer em tempo de execução. Trata-se de uma segurança a mais para o programador. Se você trabalha, por exemplo, com código que lida com estruturas muito precisas de dados, esta segurança é garantida. É um inteiro de 32 bits que você precisa? O compilador garantirá que apenas este tipo de variável seja acessada pelo seu código e, caso algum trecho lhe envie um valor diferente, o erro será detectado para você.

A tipagem estática traz essa sensação segurança ao programador, mas é importante salientar que usei aqui a palavra "sensação" por uma boa razão: a tipagem estática apenas consegue detectar os erros que um programador de QI extremamente baixo cometeria. Observe o código a seguir escrito em Java:

```
public class Calculadora {
    public int soma(int x, int y) {
        return x + y;
    }

    public static void main(String args[]) {
        Calculadora calc = new Calculadora();
        int x = java.lang.Integer.MAX_VALUE;
        int y = 34;
```

```
        System.out.println(calc.soma(x,y));
    }
}
```

O código compilará perfeitamente (experimente). Em teoria, a tipagem estática nos garante que passamos valores válidos para nossa função `soma`, mas qual será o valor impresso na saída do programa? Será positivo ou negativo, levando-se em consideração que os parâmetros passados eram ambos positivos? Resposta: `-2147483615`. Sei que este é um exemplo bastante simplista e que apenas um tolo escreveria, mas é altíssima a probabilidade de ser cometido por **você** em um teste caso lhe peçam para implementar uma busca binária [32].

O exemplo serve apenas para desbancar um mito comum relativo à tipagem dinâmica: o de que sua cobertura de testes precisa ser bem maior. A esmagadora maioria dos erros detectados pela tipagem estática são, na realidade, erros de sanidade que raríssimas vezes passam batidos pelos programadores. Tanto tipagem dinâmica quanto estática requerem praticamente a mesma quantidade de testes.

Já com relação à tipagem dinâmica, sua principal vantagem é a concisão do código escrito. Lendo atentamente o trecho a seguir, fica claro que há uma "certa" redundância no que estamos escrevendo:

```
/*
   Eu sei que é um item, preciso repetir duas vezes?
*/
Item item = new Item()
```

Repare como fica bem mais claro no exemplo a seguir:

```
def item = new Item()
```

O leitor deve levar em consideração o fato de que códigos que alteram repetidas vezes o tipo de uma variável são extremamente raros. Na esmagadora maioria dos casos ao escrevermos nossos programas não precisamos ou desejamos fazer isto. Um bom exemplo de concisão obtida com a tipagem dinâmica pode ser visto na função a seguir:

```
def soma (a, b) {
    a + b
}
```

Observe que não definimos os tipos dos parâmetros a e b, sendo assim, qualquer uma das condições a seguir é válida:

- inteiro, inteiro
- inteiro, long
- long, inteiro
- long, long
- BigDecimal, inteiro
- e por aí vai

Se estivéssemos usando tipagem estática, seria necessário escrever uma variação daquela função para cada uma das possibilidades de combinação de tipos numéricos em nossa linguagem. Já imaginou? Eu sim:

```
int soma (int a, int b) { a + b }
long soma (long a, long b) { a + b }
// imagine o restante e agradeça pelo fato
// deste exemplo ser escrito em Groovy e não Java!
```

A grande desvantagem da tipagem dinâmica diz respeito ao tempo de execução, visto que precisamos descobrir o tipo de cada parâmetro ou variável enquanto o programa é executado. No entanto, deve ser levado em consideração que aqui estamos falando da JVM. E a JVM possui o recurso da compilação *just in time*, que otimiza nossos sistemas durante sua execução. Sendo assim, as primeiras execuções de sistemas que possuam tipagem dinâmica realmente serão mais lentas, porém se tornarão mais rápidas conforme otimizações forem descobertas pelo compilador. Por exemplo, se for detectado que para a função `soma` exposta no penúltimo exemplo sempre estão sendo passados valores inteiros, otimizações poderão ser executadas. Mas, claro, dificilmente chegaremos ao desempenho de um código estaticamente tipado.

Sabem o que é mais legal em Groovy? É que podemos ter o melhor dos dois mundos. Tipagem estática quando necessária e dinâmica quando desejável.

Aplicando tipagem estática

Aplicar tipagem estática em Groovy é muito simples. Basta que ao declararmos nossas variáveis já forneçamos qual o tipo que esperamos para ela, como nos exemplos que seguem:

```
String queroUmaString
int queroUmInteiro
Item queroItem
```

Se quiser, você também pode reforçar ainda mais a aplicação da tipagem estática em seu código Groovy com a anotação `@CompileStatic`, que pode ser aplicada tanto em métodos quanto em definições de classe. O resultado de sua aplicação é simples: a verificação de tipos será feita em **tempo de compilação** no código anotado, proporcionando ao desenvolvedor código com desempenho superior. Quer ver alguns exemplos?

```
/*
    Aplicando apenas em um método
*/
class Calculadora {
    @CompileStatic
    int soma(int x, int y) {
        x + y
    }
}

/*
    Aplicando na classe inteira
*/
@Compilestatic
class Calculadora {
    // conteúdo omitido
}
```

Aplicando tipagem dinâmica

Basta fazer o oposto: não declare os tipos dos seus parâmetros ou retorno de funções além de suas variáveis. :)

Princípio do pato

A tipagem dinâmica mostra o aumento de produtividade do desenvolvedor principalmente quando precisamos lidar com objetos. A propósito, há um detalhe da linguagem Groovy que não mencionei até este momento: ao contrário do Java, não há tipos primitivos. Repare no código a seguir:

```
int i = 3
/*
    Em Java a instrução abaixo sequer seria
    compilada
*/
print i.getClass() //Imprime java.lang.Integer
```

Tudo em Groovy é um objeto, o que caracteriza a linguagem como sendo puramente orientada a objetos. Sendo assim, se declaramos um tipo primitivo em Groovy, este é automaticamente convertido para uma classe *wrapper*, como `java.lang.Integer`, `java.lang.Boolean` etc. Esta razão junto ao princípio do pato que irei expor a seguir são os pilares que possibilitam o aspecto dinâmico do compilador.

Mas o que é o princípio do pato?

- Anda como um pato?
- Fala como um pato?
- Voa como um pato?
- Nada como um pato?
- É... temos um pato!

Fig. 3.1: Talvez seja pato e coelho

Em código-fonte, este princípio fica ainda mais evidente, observe o exemplo:

```
class Galinha {
    String grite() {"Có!"}
}

class Pato {
    def grite() {"Quá!!!"}
}

def fale(bicho) {
    println bicho.grite()
```

```
}

fale(new Galinha())
fale(new Pato())
// Imprimirá
// Có!
// Quá!!!
```

Veja que declaramos duas classes que possuem um método com o mesmo nome e mesmo número de parâmetros, o método `grite`. Ainda mais interessante, note que não há uma interface em comum entre as classes `Galinha` e `Pato`, e que a função `fale` recebe como parâmetro um valor cujo tipo é indefinido.

Nossa função `fale` internamente apenas espera que o objeto passado como parâmetro possua a função `grite()` e que esta me retorne algum valor que será, em seguida, impresso. Tanto a classe `Galinha` quanto `Pato` possuem algo similar. Eles andam, falam, nadam e voam como patos: para nossa função são portanto... **patos**.

Trata-se de código muito menor e fácil de ser compreendido. Podemos compará-lo com a versão feita em Java aplicando os conceitos de tipagem estática:

```
public interface Grite {
    String grite();
}

Pato implements Grite {
    public String grite() {
        return "Quá!!!";
    }
}

class Galinha implements Grite {
    public String grite() {
        return "Có!";
    }
}
```

```
class Torturador {
    void fale(Grite grite) {
        System.out.println(grite.grite());
    }
}
```

Este princípio vai além em Groovy. Ele é aplicado também nos tipos numéricos e strings. Por isso, não precisamos fazer coerção de tipos quando somamos um `long` e um `int` em Groovy, ou mesmo um `long` e um `BigDecimal` ou qualquer outro tipo numérico. Esperamos que todas essas classes implementem as operações aritméticas da mesma forma, então, por que não manter as coisas simples?

3.4 Closures

Já que estamos falando de tipos de dados, somos obrigados a tratar aqui de um tipo muito especial contido em Groovy desde sua primeira versão e que só surgiu na versão 8 do Java: as closures. Pense nelas como um tipo de dados qualquer, porém com uma diferença sutil: em vez de armazenar dados, armazena código executável. Esta sutilidade torna Groovy uma linguagem perfeitamente apta à aplicação do paradigma funcional de programação.

Mas antes de falar de programação funcional, precisamos mostrar como declaramos uma closure. No exemplo a seguir declaro uma closure que, ao ser executada, imprimirá "Olá mundo":

```
def olaMundo = {
    println "Olá mundo"
}
```

Uma vez declarada, executar uma closure é bastante simples: você o fará exatamente como faria se fosse chamar uma função:

```
olaMundo()
// imprime "Olá mundo"
```

A closure é uma função. Funções sem parâmetros têm sua aplicação bastante limitada. Posso incluir parâmetros em uma closure, certo? Sim, na verdade, sempre há pelo menos um parâmetro. Observe o código:

```
def duplique = {
    it * 2
}
duplique(1) // retornará 2
```

Toda closure possui um parâmetro implícito chamado `it`. Se quiser, você pode trocar o nome dele também.

```
def duplique = {valor ->
    valor * 2
}
duplique(1) // retornará 2
```

A sintaxe é bastante simples: basta incluir os nomes dos parâmetros antes de `->`. E, sim, você pode também aplicar tipagem estática nestes atributos.

```
def duplique = {int valor ->
    valor * 2
}
duplique(2) // retornará 4
```

E como é quando temos mais de um parâmetro?

```
def soma = {x, y ->
    x + Y
}
soma(2,3) // retornará 5
```

Closure como parâmetro

Até este momento, as closures não passam de uma curiosidade interessante ou, quem sabe, no máximo uma maneira diferente de se declarar funções. O recurso fica realmente interessante quando nos lembramos que closures são variáveis. Se são variáveis, posso passá-las como parâmetros para outras funções. Que tal implementar uma calculadora usando closures?

```
def soma = {x,y -> x + y}

def subtracao = {x,y -> x - y}
```

```
def multiplicacao = {x,y -> x * y}

def divisao = {x,y -> x / y}

// A mágica acontece
def calculadora(closure, x, y) {
    closure(x,y)
}
```

Declaramos quatro closures (uma para cada operação matemática elementar) e uma função que recebe como parâmetro uma closure e os parâmetros x e y. Tudo o que esta função faz é executar a closure com os parâmetros que lhe passamos. Que tal vermos isto na prática?

```
calculadora soma, 2, 4 // retorna 6
calculadora subtracao,  4, 2 // retorna 2
calculadora multiplicacao, 3, 6 // retorna 18
calculadora divisao, 6, 3 // retorna 2
```

Este é um exemplo bastante simples da aplicação desta técnica, no entanto, o leitor observará o poder das closures quando aplicados na escrita de bibliotecas, tal como a linguagem aplica na API Collections do Java. Observe o código a seguir:

```
def lista = ["z","38", "d", "a", "3k"]
lista.findAll {
    it.length() > 1
}
// resultado: ["38", "3k"]
```

Várias funções são incluídas na API Collections. Entre elas, encontra-se a função `findAll`, que recebe como parâmetro uma closure. Espera-se que esta closure retorne um valor booleano. Em nosso exemplo, declaramos a closure diretamente na chamada da função (uma closure anônima), que apenas verifica se o valor que lhe é passado possui um texto maior do que um caractere.

Veremos estas aplicações em toda a API Java que o Groovy toca, além de, claro, extensivamente em Grails.

Escopo

Há outro aspecto interessante nas closures: elas herdam o escopo de onde foram declaradas. Para explicar melhor este conceito, observe o script a seguir:

```
class Pessoa {
    String nome

    def apresenteSe = {
        println "Olá. Meu nome é $nome"
    }
}

class Animal {
    def closure
    def nome

    def fale() {
        closure()
    }
}

def pessoa = new Pessoa(nome:"Henrique")
def animal = new Animal(nome:"Cão")
animal.closure = pessoa.apresenteSe
animal.fale() //imprimirá "Olá. Meu nome é Henrique"
pessoa.nome = "Angélica"
animal.fale() //imprimirá "Olá. Meu nome é Angélica"
```

A closure sempre terá acesso a todas as variáveis independente da visibilidade da classe na qual foi declarada. Repare que na classe `Animal` também há um atributo chamado `nome`, cujo valor é *"Cão*

. No entanto, a função `fale` ao executar a closure apenas imprime os atributos da classe `Pessoa`.

this, owner e delegate

No corpo de cada closure há três atributos que sempre estarão presentes:

this, owner e delegate. É interessante que você saiba como tirar proveito deles.

Os atributos this e owner, na prática, são a mesma coisa. Ambos referenciam a classe na qual a closure foi referenciada. Vamos modificar um pouco o último exemplo para expor o uso dos atributos owner e this.

```
class Animal {
    def fale() {
        println closure.owner
        closure()
    }
}
def pessoa = new Pessoa(nome:"Henrique")
def animal = new Animal(nome:"Cão")
animal.fale()
// Imprimirá algo como
// Pessoa@5d38398
// Olá. Meu nome é Henrique
```

O atributo this não é acessível externamente. Ao contrário de owner, delegate pode ser alterado em tempo de execução. Sendo assim, podemos modificar um pouco mais o último exemplo para expor seu uso:

```
class Pessoa {
    String nome

    def apresenteSe = {
        println "Olá. Meu nome é $nome e meu delegate ${delegate.nome}"
    }
}

class Animal {
    def closure
    def nome

    def fale() {
        closure.delegate = this
        closure()
    }
```

```
}


def pessoa = new Pessoa(nome:"Henrique")
def animal = new Animal(nome:"Cão")
animal.closure = pessoa.apresenteSe
animal.fale() //imprimirá "Olá. Meu nome é Henrique e meu delegate Cão"
```

O atributo `delegate` pode ser alterado para qualquer objeto do sistema. Em nosso exemplo, alteramo-lo para que correspondesse àquele responsável por executar a nossa closure, fornecendo a este acesso a todos os atributos do objeto chamante.

E a programação funcional?

No início desta seção, mencionei que as closures permitem que Groovy se torne uma excelente linguagem para a aplicação do paradigma funcional, lembra? Basta reler esta seção e observar que com closures conseguimos atingir os principais aspectos que caracterizam este paradigma:

- Funções de primeira classe: são funções que você pode passar como parâmetro para outras funções ou que podem ser o resultado de uma função.
- Funções puras: aquelas que não alteram o estado do sistema. Elas apenas retornam um resultado a partir dos parâmetros que receberam como entrada.
- Recursão: nossas closures podem ser recursivas.

3.5 Metaprogramação

Metaprogramação é um daqueles conceitos que você pode passar toda a sua vida profissional sem conhecer mas, uma vez conhecido, se perguntará como pode passar tanto tempo ignorando-o. Em sua essência, estamos falando da habilidade de um programa de manipular ou mesmo criar outros programas.

É um recurso poderosíssimo e que nos abre uma imensa gama de possibilidades na escrita de soluções mais simples e elegantes. Caso o leitor esteja ansioso para lidar com Grails, peço para que não pule esta seção pois ela facilitará enormemente sua compreensão a respeito do funcionamento deste framework.

O que mostraremos aqui pode ser obtido com Java também, porém de uma forma extremamente mais trabalhosa através do uso de bibliotecas que manipulem *bytecode* (como ASM ou CGLib). A grande vantagem do Groovy é que obtemos o mesmo resultado usando apenas a linguagem e nada mais.

Incluindo novos métodos e atributos em classes preexistentes

Imagine que em um de nossos sistemas precisemos o tempo inteiro incluir uma string entre colchetes. Uma solução simples poderia ser criarmos uma nova função tal como a exposta a seguir:

```
String entreColchetes(String valor) {
    "[ $valor ]"
}
```

Talvez pudéssemos declará-la como um membro estático de uma classe e, em seguida, apenas referenciar aquela classe em todos os pontos do nosso sistema. Outra solução seria modificar o código-fonte da classe `String` do Java, mas esta não é uma solução adequada. Além disto, estamos lidando com uma classe final e que não pode ser modificada ou estendida, certo? Se você falar em Java, sim. Já no mundo Groovy, não. Observe o código:

```
String.metaClass.entreColchetes = {
    "[ $delegate ]"
}
"Groovy rocks".entreColchetes()
// Resultado: "[ Groovy rocks ]"
```

Você leu certo: acabei de incluir um novo método na classe `java.lang.String` em tempo de execução. Quando programamos em Groovy, em todas as classes do sistema, sempre é incluído o atributo `metaClass`, que funciona exatamente como um mapa. O que fi-

zemos foi essencialmente incluir uma nova chave neste mapa chamada `entreColchetes` que recebe como parâmetro uma closure.

No interior desta closure, estamos referenciando o objeto `delegate` que apontará, neste caso, para a instância da classe que acabamos de modificar. Por isso, conseguimos colocar o conteúdo entre colchetes apenas interpolando strings.

Ainda mais interessante, você pode também adicionar novos comportamentos não a uma classe, mas sim a uma instância específica apenas. Veja o exemplo a seguir:

```
class Pessoa {
    String nome
    void digaOi() {
        println "Olá, sou $nome"
    }
}
def pessoaNormal = new Pessoa(nome:"Henrique")
// Imprimirá "Olá, sou Henrique"
pessoaNormal.digaOi()
def pessoaAlterada = new Pessoa(nome:"Juca")
pessoaAlterada.metaClass.digaOi = {
    println "Fui alterado!"
}
// Imprimirá "Fui alterado!"
pessoaAlterada.digaOi();
// Imprimirá "Olá, sou Henrique"
pessoaNormal.digaOi()
```

Repare que podemos acessar o atributo `metaClass` em apenas uma instância se quisermos, mantendo inalterada a classe original. Este é um comportamento interessante quando desejamos alterar nossos objetos apenas em casos mais específicos.

Também é possível adicionar métodos a uma classe ou instância. Basta que adicionemos ao atributo `metaClass` que não tenha como valor uma closure:

```
class Pessoa {
    // Repare que não há atributo algum
```

```
}
Pessoa.metaClass.nome = "Henrique"
de pessoa = new Pessoa()
// Yeap: Groovy já cria automaticamente o getter e setter para nós!
// imprimirá "Henrique"
println pessoa.getNome()
// E o setter também!
pessoa.nome = "Angélica"
// Imprimirá "Angélica"
println pessoa.nome
```

Interceptando chamadas a métodos

Se adicionar métodos ou atributos a classes já existentes parece uma ideia interessante, imagine poder interceptar a chamada destes métodos. Groovy nos permite fazer isto de uma forma bastante simples usando a função `invokeMethod` da nossa classe. Toda classe no contexto Groovy a possui, e no exemplo a seguir podemos ver como proceder:

```
class Pessoa {
    String nome
    void imprima() {
        println "Meu nome é $nome"
    }
}

// Aqui o truque é preparado
Pessoa.metaClass.invokeMethod = {String name, args ->
    println "Vou chamar $name"
    def metodo = delegate.metaClass.getMetaMethod(name, args)
    metodo.invoke(delegate, args)
    println "Chamei o método"
}

// Aqui a mágica ocorre
def pessoa = new Pessoa(nome:"Henrique")
pessoa.imprima()
// Eis a saída
```

```
// Vou chamar imprima
// Meu nome é Henrique
// Chamei o método
```

No exemplo, usei o mesmo atributo `metaClass` que vimos no tópico anterior. Repare que sobrescrevo o método `invokeMethod` da nossa classe passando uma closure que espera dois parâmetros: o primeiro obrigatoriamente deve ter seu tipo explicitamente definido (diz respeito ao nome da função/método a ser interceptado) enquanto o segundo identifica os parâmetros que serão passados para o método a ser interceptado.

No interior de nossa versão `invokeMethod`, vamos fazer duas coisas: obter o método em nossa classe chamando a função `getMethod` do atributo `metaClass` e invocá-lo. Mas antes disto vamos informar a nosso usuário que interceptamos o método e ao final diremos que executamos o método interceptado.

Bom, mas por que você iria querer interceptar métodos em uma classe? Dou-lhe uma boa razão: para tirar proveito de outro paradigma de programação: a *Programação Orientada a Aspectos* (*Aspect Oriented Programming* AOP). Este é um momento no qual podemos ver o poder do Groovy: sabemos que estamos lidando com uma linguagem extremamente poderosa quando esta nos permite acessar outros paradigmas de programação de uma forma simples. Em Java, para tirarmos proveito da AOP precisamos de frameworks como Spring ou AspectJ. Em Groovy... precisamos apenas do Groovy. Vamos agora para um exemplo mais concreto: como funciona um framework de segurança?

Um framework de segurança apenas intercepta as chamadas a métodos: caso o usuário possua acesso àquele método, nós o executamos. Caso contrário, bloqueamos o acesso. Vamos a um exemplo rápido?

```
class Integrador {
    def execute() {
        // ignore o interior
        // apenas para fins didáticos
    }
}
```

```
// Nosso código de segurança
Integrador.metaClass.invokeMethod = {String name, args ->
    if (testeAcessoUsuario()) {
        // Usuário possuí acesso
        return delegate.metaClass
                .getMetaMethod(name, args)
                .invoke(delegate, args)
    } else {
        // Usuário não possuí acesso
        // dispare uma exceção
        throw new RuntimeException("Acesso negado")
    }
}
```

Neste exemplo não nos interessa saber como o método `testeAcessoUsuario` funciona: o que interessa é que ele retornará `true` caso quem chama a função possua acesso a esta. Observe algo interessante: estamos retornando o valor do método chamado. Sim: através da interceptação de métodos podemos substituir o valor retornado. Veja o exemplo:

```
class Calculadora {
    def soma(x, y) {
        x + y
    }
}

Calculadora.metaClass.invokeMethod = {String name, args ->
    def method = delegate.metaClass.getMetaMethod(name, args)
    return 2 * method.invoke(delegate, args)
}

// Retornará 10, e não 5
new Calculadora.soma(2,3)
```

Estamos lidando aqui com um recurso extremamente poderoso e perigoso. Mas podemos ir além: se é possível interceptar a chamada de métodos, será que também podemos interceptar a execução de métodos que **não existem**? Sim, podemos, e o código a seguir nos ensina como fazer isto.

```
class Calculadora {
    def soma(x,y) {x + y}
}

Calculadora.metaClass.methodMissing = {String name, args ->
    println "Não conheço este método $name"
}
// Será impresso "Não conheço o método subtracao"
new Calculadora().subtracao(3,4)
```

Basta sobrescrever o método `methodMissing` na nossa classe alvo, exatamente como faríamos se estivéssemos interceptando a chamada de funções. Quer um exemplo de aplicação deste recurso? Os *finders dinâmicos* do Grails que veremos mais à frente neste livro são implementados através dele.

O mesmo princípio também pode ser aplicado a propriedades. Basta sobrescrever a função `propertyMissing` tal como no exemplo a seguir:

```
Calculadora.metaClass.propertyMissing = {String name ->
    println "A propriedade $name não existe"
}
// Imprimirá "A propriedade nome não existe"
new Calculadora().nome
```

Igualmente simples. Você pode tirar proveito dessa funcionalidade para marcar as classes da sua aplicação que ainda não tenham sido processadas de alguma maneira. Imagine que você queira verificar se uma classe de domínio da sua aplicação já foi processada incluindo métodos de persistência. Você poderia simplesmente verificar a ausência de uma propriedade nesta classe.

3.6 Invocação dinâmica de métodos

Um último recurso interessante do Groovy: invocação dinâmica de métodos ou propriedades. Tentar descrevê-lo seria muito chato, sendo assim vou expô-lo a você usando código:

```
class Calculadora {
    def soma(x,y) {x+y}
    def subtracao(x,y) {x-y}
```

```
    def multiplicacao(x,y) {x*y}
    def divisao(x,y) {x/y}
}

def executeCalculo(String nomeOperacao, x, y) {
    // Repare como executo o método pelo nome
    new Calculadora()."$nomeOperacao"(x,y)
}
// Resultado: 7
executeCalculo("soma", 3, 4)
```

Podemos invocar um método em nossos objetos tanto a partir do modo tradicional como a partir da representação do seu nome como uma string! Há diversas aplicações para esse tipo de recurso. A principal que menciono nesta seção seria na criação de uma DSL (*Domain Specific Language* Linguagem Específica de Contexto). Imagine uma linguagem similar à que exponho a seguir:

```
soma 3 4
subtracao 34 4
divisao 4 2
```

É fácil implementar um parseador que leia cada uma das linhas expostas nesse arquivo, separe os componentes de cada operação usando o caractere de espaço e, em seguida, envie-os repetidamente para a função `executeCalculo` que expus no exemplo desta seção. E este é apenas um exemplo tolo. Conforme você for adquirindo mais prática com Groovy, vai encontrar inúmeras outras possibilidades em seus sistemas. Especialmente quando estiver trabalhando com Grails.

3.7 Concluindo

Acredito que, finalizado este capítulo, caso seu background seja Java, a pergunta que fiz no capítulo anterior "para que outra linguagem de programação?" tenha sido plenamente respondida. Como pôde ser visto neste capítulo, estamos diante de não apenas "mais uma linguagem de programação", mas sim "uma **poderosa** alternativa ao Java", que nos permite tirar máximo

proveito de conceitos como metaprogramação, AOP, invocação dinâmica de métodos e tantas outras coisas de uma forma extremamente simples e pragmática.

Mais do que isto, nos treinamentos que ofereço sobre Grails observo que diversos alunos que pulam o Groovy e caem direto no framework têm a impressão de que estão lidando com funcionalidades que surgem "automagicamente". Como deve ter percebido, não surgem como mágica, mas sim como consequência da linguagem na qual o framework foi escrito.

Você está pronto para que possamos começar a falar sobre Grails agora. Siga em frente e comece a criar aplicações web extremamente poderosas com altíssima produtividade.

Capítulo 4

Precisamos falar sobre Grails

No FISL (Fórum Internacional de Software Livre) de 2005 que ocorreu em Porto Alegre, nós, desenvolvedores Java brasileiros, levamos o mais delicioso tapa na cara que poderíamos tomar. Graças a uma apresentação de 16 minutos de um sujeito chamado David Heinemeier Hansson (o tal do DHH), acordamos para o fato de que o modo com o qual estávamos acostumados a desenvolver nossos sistemas era lento, pouco produtivo, repetitivo e muito burocrático. Sua apresentação chamada *How to build a blog engine in 15 minutes with Ruby on Rails* [13] (*Como construir um motor de blog em 15 minutos com Ruby on Rails*) foi logo em seguida publicada no YouTube e, do dia para a noite, muitos de nós (me incluo) se sentiram envelhecidos no mínimo uns 50 anos.

Foi uma apresentação cheia de falhas, muitos "Ops!" e uma aplicação espartana para dizer o mínimo, mas isso não nos interessava; o que nos fazia babar era a execução de conceitos sobre os quais até então ouvíamos dizer que

eram legais mas nunca haviam se mostrado de forma tão explícita. Conceitos como DRY (*Don't Repeat Yourself*, *Não se repita*), convenções sobre configuração, testabilidade, KISS (*Keep it Simple Stupid*, *Mantenha isto simples estúpido*), *scaffolding*. Aquele gringo ia alterando suas classes de domínio e, como que por mágica, todas as páginas eram geradas automaticamente na nossa frente.

Após aquela apresentação, programar em JSF 1.1, que era a última palavra em desenvolvimento web na plataforma Java EE (em 2006 sairia a versão 1.2), se tornara um teste de paciência e a plataforma Java EE se mostrava como uma das mais tediosas. É engraçado pensar que até alguns dias antes era algo *realmente* legal. Uma pena: Java estava acabado, era o fim.

No exterior, Ruby on Rails já era bastante conhecido e naquele mesmo ano de 2005 já surgia uma alternativa real para salvar o Java EE: um novo framework baseado em Groovy com o sugestivo nome *Groovy on Rails*. Agora tínhamos algo que estava na mesma altura que o Ruby on Rails. Podíamos fazer tudo aquilo que o DHH nos mostrou, só que tendo acesso direto a todo o nosso código Java que lutamos tanto para "tediosamente" escrever ao longo de 8 anos desde que Java tomou o mundo de assalto. Logo depois, a pedido do nosso amigo DHH, o framework teve de mudar de nome para *Grails* e este tem sido seu nome desde então. Inicialmente desenvolvido por uma empresa chamada G2One, foi logo em seguida comprado pela SpringSource, que hoje faz parte da Pivotal, uma divisão da VMWare.

Grails é um framework cujo modo de trabalho é extremamente dinâmico e que acabou influenciando todos os frameworks Java que lhe sucederam de forma profunda. Não vejo outra forma de introduzi-lo a vocês que não seja da maneira mais prática possível, ou seja, como Platão o faria, dialogando.

4.1 Falando de Grails

Alguns meses atrás, Kico foi contratado como programador por uma empresa de engenharia especializada em projetos industriais chamada *DDL Engenharia*. A DDL se orgulha de uma longa trajetória construída ao longo de três décadas no planejamento e execução de projetos nas áreas de mineração, metalurgia, óleo e gás.

Como é comum ocorrer em empresas cujo foco não é desenvolvimento de software, as três décadas da DDL trouxeram um longo fardo de sistemas legados desenvolvidos nos mais variados ambientes de desenvolvimento, como por exemplo Visual Basic (pré .NET), Pascal, C, Fortran. Mais recentemente, iniciou-se a tendência de desenvolver aplicações usando a plataforma Java EE na esperança de que se tratasse de uma plataforma sólida para a próxima década.

Um dia, Kico se encontrou com Guto, responsável pelo setor de suprimentos da empresa na sala do café e no decorrer da conversa relatou seu drama:

Kico, estou perdido cara. Até então eu me virava muito bem com as minhas planilhas na hora de controlar as cotações que faço para nossas obras, mas de repente, DO NADA o MAIOR CLIENTE DA EMPRESA resolveu que todos os seus fornecedores devem disponibilizar online seus sistemas de cotações na internet. Perguntei se deixando minhas planilhas compartilhadas em algum site adiantaria mas eles disseram que não, que tem de ser interface web.

Ah Guto, isso é fácil de resolver: eu posso escrever uma aplicação web para você em algumas horas e depois a gente vai evoluindo a coisa conforme novas demandas vão aparecendo, o que me diz?

Kico, você está tendo uma das suas alucinações. Aconselho que pare de tomar estes chás esquisitos o quanto antes, pois já ouvi essa história inúmeras vezes: vocês vão chamar um tal de analista de requisitos para vir conversar comigo. Em seguida, este sujeito vai ficar me atazanando O DIA INTEIRO sem me deixar trabalhar e daqui a alguns meses você vai me entregar aquele negócio que não vai me atender. Nem vem com esse papinho furado pra cima de mim cara.

Guto, tudo o que você precisa fazer é me explicar o que você precisa e juntos sairemos da sua sala com uma versão inicial do sistema, o que me diz? Aposto meu PS4 com você.

Um sorriso malicioso surgia na face de Guto.

O sistema

No caminho para a sala de Guto, o problema foi exposto. O processo de cotação consistia basicamente na execução de uma pesquisa de preços. O trabalho de Guto era, dado um item para o projeto, como por exemplo uma bomba ou um motor, consultar sua lista de fornecedores em busca do preço deste produto. A cotação nada mais seria do que o preço do item em um dado dia por um fornecedor específico. Esta era entregue ao cliente, que então escolhia qual o fornecedor que melhor lhe atendesse.

Entendo, Guto: então imagino que nosso sistema possua quatro entidades. Uma categoria na qual os itens se encaixem, como por exemplo **Equipamentos** ou **Materiais**, o item propriamente dito, que pertence a essas categorias, como **Britador** e **Cimento**. Dado que a pesquisa se dá por fornecedores, teríamos também uma lista de fornecedores e, finalmente, a cotação com os campos de valor, fornecedor, item e data, certo?

É... Mas tem outro detalhe: a gente faz projetos para o exterior, e muitos itens de cotação vêm de fora, então a moeda pode variar.

Começou a complicar, hein?

Você que quis apostar, se vira.

Terminaram o sistema com cinco entidades iniciais: Categoria, Item, Cotação, Fornecedor e Moeda, o que já fornecia uma base mínima para que pudesse ser executado.

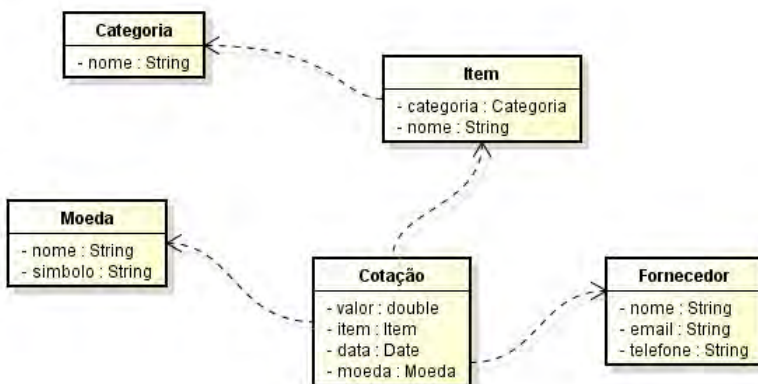



Fig. 4.1: As entidades iniciais do sistema

4.2 O que é um framework?

A DDL Engenharia possuía uma longa tradição no desenvolvimento de aplicações voltadas para o ambiente desktop. Esta é a razão pela qual Visual Basic era a linguagem predominante. O novo requisito do cliente iria trazer um impacto muito mais profundo à empresa e forçá-la a, pela primeira vez, desenvolver uma solução para um ambiente de execução inexplorado, a web.

O modo de trabalho também era diferente: até então todos estavam acostumados a um modelo de desenvolvimento no qual primeiro se desenhava o banco de dados e a interface gráfica usada para alimentá-lo em uma ferramenta RAD (*Rapid Application Development*) como o Visual Basic. Caso houvesse funções que fossem compartilhadas por mais de um formulário, elas eram incluídas em um módulo compartilhado por toda a aplicação e assim ia-se desenvolvendo o sistema, um formulário e uma tabela por vez.

A carta que Kico trazia em sua manga era Grails: um framework focado no desenvolvimento de aplicações web. Framework, uma palavra tão comum entre desenvolvedores Java, nem sempre é um conceito tão autoevidente quanto imaginamos.

Para resolver este problema, usaremos Grails, Guto. Um framework para desenvolvimento web baseado na linguagem Groovy. Ele gera aplicações Java EE e, portanto, pode ser executado tanto em um servidor simples quanto de aplicações corporativas Java EE.

Huhn? Framework? O que é isto, uma biblioteca como aquelas que usamos nas nossas aplicações correntes ou um daqueles componentes? Palavrinha chique para me enrolar hein?

Nem um nem outro, apesar de possuir atributos de ambos. Quando desenvolvemos uma aplicação com Visual Basic, por exemplo, usamos bibliotecas e componentes que executam suas tarefas somente quando nós programaticamente os adicionamos, certo?

Correto: eu simplesmente chamo minhas funções quando preciso delas. O que você quer dizer com isto é que no framework é diferente, como funciona isto?

No caso do framework, ocorre o contrário: não somos nós que escrevemos o código que aciona nossos componentes. Nosso papel é escrever o

componente que será acionado pelo framework. Por exemplo: ao invés de escrever o código que lida com o momento no qual nosso servidor recebe uma requisição, nós iremos desenvolver o código que é chamado pelo framework quando uma requisição é recebida (são os tais controladores que veremos mais à frente).

O que ocorre aqui é a chamada inversão de controle, que na prática aplica o princípio de Hollywood. Quando amadores procuram produtores de cinema em busca de um papel, apresentam seus currículos com suas capacidades para que sejam avaliadas. O que os produtores fazem é lhes dizer para que não lhes perturbem o tempo inteiro perguntando se há um papel disponível. Quando (se) houver, um ator será chamado imediatamente. **"Não nos procure, deixe que procuraremos você**

.

É o framework que chama nossos componentes, a situação se inverteu: não precisamos mais nos preocupar com a lógica por trás do acionamento destes. Na prática, um framework é como uma aplicação semipronta: todo o código de infraestrutura já se encontra implementado por alguém que passou meses projetando a melhor maneira de lidar com aquela situação (no nosso caso, a grosso modo, lidar com requisições web).

Sei que o conceito provavelmente não está muito claro neste momento, mas ficará quando terminarmos a primeira versão do nosso sistema de cotações.

4.3 Instalando o Grails

Há duas maneiras de se instalar o Grails. Neste capítulo, trataremos da mais comum: baixar a distribuição mais recente, descompactá-la e, em seguida, alterar algumas variáveis de ambiente. A outra maneira é através de alguma IDE, que normalmente já vem com o framework embutido entre suas dependências e portanto tornam o processo de instalação desnecessário.

O primeiro passo na instalação do Grails é baixar a última versão no site oficial do projeto, http://www.grails.org. Durante a escrita deste livro, foi usada a versão 2.4.4. As distribuições do Grails vêm no formato `zip`. Tudo o que o usuário deve fazer é descompactar este arquivo em um diretório de sua

preferência. No nosso caso o conteúdo foi descompactado em um diretório chamado `C:\grails\grails-2.4.4`.

Não é preciso ter o Groovy instalado em seu computador para trabalhar com Grails: a linguagem já vem embutida na distribuição do framework. O único requisito é ter instalado em seu computador o JDK versão 1.6 ou posterior. Dê preferência sempre à última versão do JDK, pois pode haver plugins que dependam de recursos do Java 7. Falaremos disso adiante.

É necessário alterar algumas variáveis de ambiente. Este procedimento varia de acordo com seu sistema operacional. Vamos expor aqui como executar essa tarefa nos três sistemas mais comuns: Windows, Linux e Mac OS X.

Variáveis de ambiente no Windows

Abra a janela principal do Windows Explorer e busque pelo ícone "Meu Computador": encontrando-o, clique com o botão direito do mouse sobre este e, em seguida, sobre a opção "Propriedades". Na imagem a seguir podemos ver um exemplo de como encontrá-lo no Windows 8.

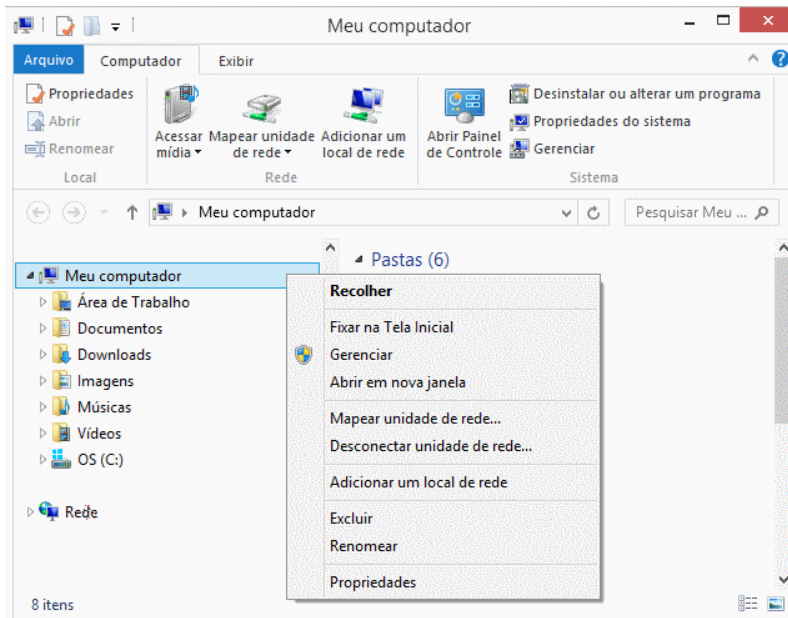



Fig. 4.2: O ícone 'Meu Computador' do Windows 8

Surgirá uma janela expondo informações sobre o seu sistema. Clique sobre a opção "Configurações avançadas do sistema" para expor a janela "Propriedades do Sistema" tal como na imagem a seguir:

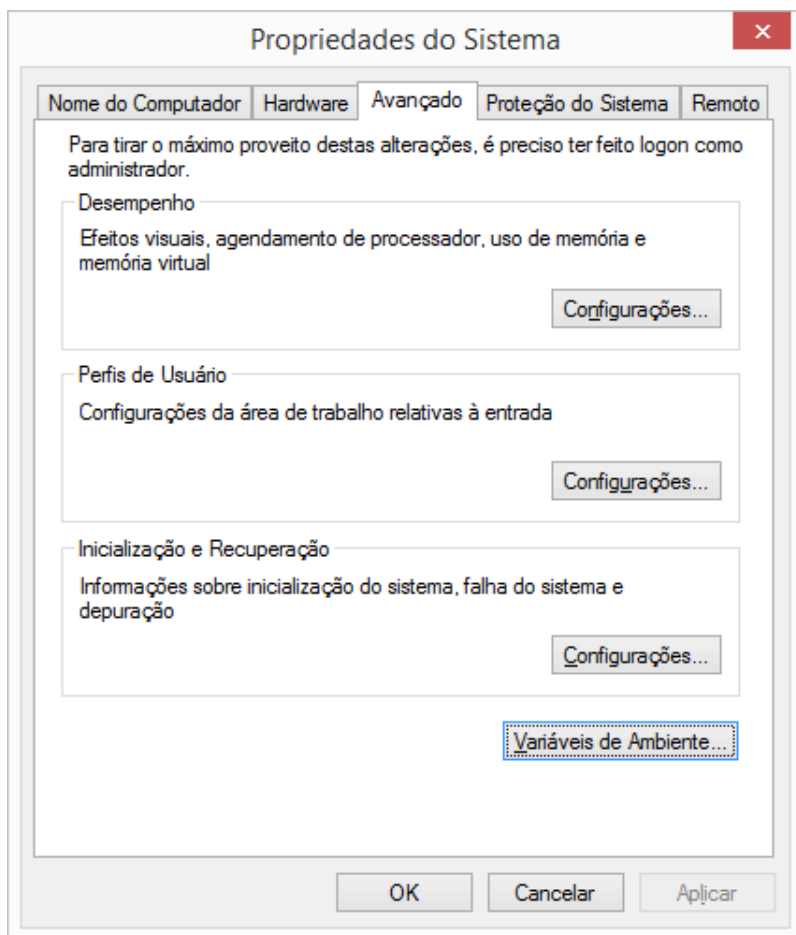



Fig. 4.3: Propriedades do sistema no Windows 8

Clique no botão "Variáveis de Ambiente". Finalmente, poderemos agora fornecer ao sistema as variáveis de ambiente que possibilitarão a execução correta do Grails.

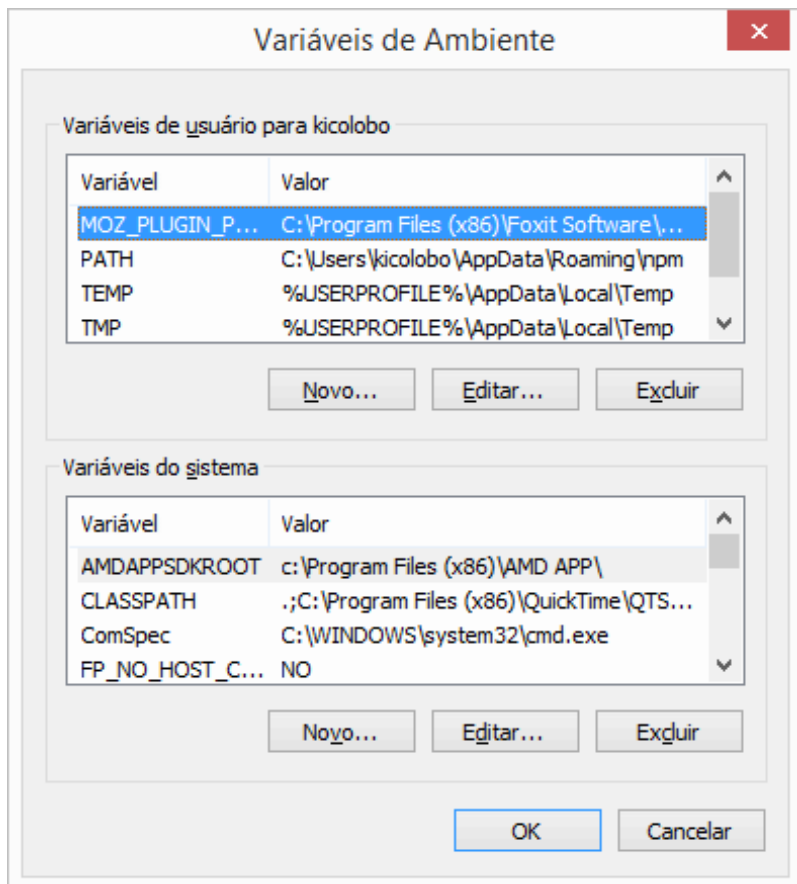



Fig. 4.4: Variáveis de ambiente do Windows 8

Variáveis de ambiente no Windows podem ser incluídas em dois grupos: apenas para o usuário corrente ou para o sistema, o que as tornarão disponíveis para todos os usuários cadastrados em seu computador. Neste exemplo, iremos incluí-las apenas no usuário corrente.

A primeira variável de ambiente se chama `GRAILS_HOME`, e seu valor deve corresponder ao diretório no qual você descompactou sua distribuição do Grails.

Precisamos verificar se está presente em seu sistema a variável de sistemas `JAVA_HOME`. No caso do Windows, é necessário checar a sua presença

nas duas caixas de seleção: "Variáveis de ambiente do usuário corrente" e "Variáveis do sistema". Certifique-se de que, caso exista, ela tenha como valor o diretório no qual o JDK está instalado. Se não existir, basta incluir o valor na listagem de variáveis de ambiente do usuário corrente.

Para terminar, precisamos alterar a variável `PATH`. Ela é responsável por tornar disponíveis à interface de linha de comando todos os arquivos executáveis que se encontram na lista de diretórios separada por ponto e vírgula em seu valor. Certifique-se de que o diretório `bin` dos valores `JAVA_HOME` e `GROOVY_HOME` encontram-se presentes. Caso não estejam, basta incluir o conteúdo a seguir neste valor: `%JAVA_HOME%\bin;%GRAILS_HOME%\bin`.

Testar sua instalação do Grails é simples: inicie o prompt de comando e em seguida execute o comando `grails`. Se tudo estiver correto, você verá uma saída similar à da imagem a seguir:

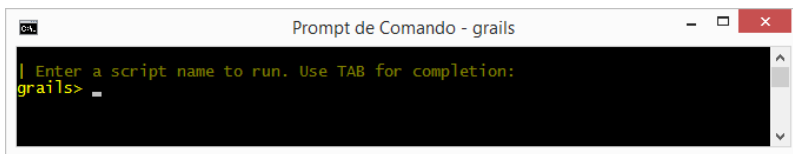



Fig. 4.5: Grails instalado corretamente

Linux e Mac OS X

A vida é muito mais fácil para usuários de sistemas Linux e Mac OS X. Tudo o que você precisa fazer é acrescentar o script a seguir ao final do arquivo `.bashrc` (no caso do Linux) ou `.bash_profile` (no caso do Mac OS X), que se encontram no diretório `home` do usuário corrente:

```
export JAVA_HOME=[o diretório da sua instalação do JDK]
export GRAILS_HOME=[o diretório da sua distribuição do Grails]
export PATH=$JAVA_HOME/bin:$GRAILS_HOME/bin
```

Também é importante que você se certifique de que os scripts presentes no diretório `bin` da sua distribuição Grails possuam permissão para execução. Feito isto, basta executar o comando `grails` na sua interface de linha de

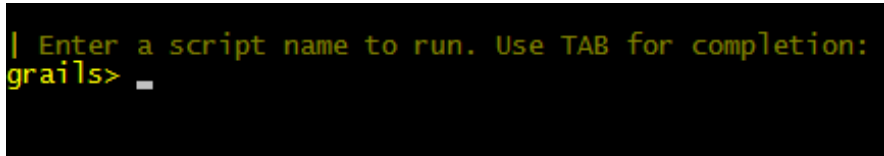
comando. Se tudo der certo, uma saída similar à exposta na imagem a seguir surgirá:

```
| Enter a script name to run. Use TAB for completion:
grails> _
```

Fig. 4.6: Grails instalado corretamente no Linux/Mac OS X

4.4 Criando a aplicação

Com o Grails instalado, nossos personagens iniciam o desenvolvimento da aplicação, batizada por ambos de *Controle de Cotações (ConCot)*.

Bom Guto, para começar o desenvolvimento vamos criar um novo projeto com a ferramenta de linha de comando do Grails. Tudo o que precisamos fazer é executar o comando `grails create-app concot`. Executado o comando, o diretório `concot` é gerado.

Programação por convenção e a estrutura de diretórios do Grails

O primeiro ponto que chamou a atenção de Guto foi o conteúdo do diretório `concot`. Dado o que havia ouvido falar a respeito da plataforma Java, imediatamente disparou contra Kico:

Pela quantidade de arquivos gerados por este comando, isso quer dizer que vamos gastar um tempo imenso editando arquivos `XML`, certo? Será que conseguiremos terminar este trabalho antes de o mês acabar (estamos no dia 10)?

Na realidade, é o contrário: dificilmente precisaremos tocar em arquivos de configuração e, diga-se de passagem, não são `XML`. Isto porque o Grails se baseia no conceito de *programação por convenção*, em inglês chamada de *convention over configuration* ou *coding by convention*.

Esse modelo de desenvolvimento busca reduzir o número de escolhas que a equipe de programação precisa fazer durante a evolução do seu projeto. São convenções adotadas pelo framework que facilitam a compreensão do sistema

e minimizam a quantidade de configuração, os famigerados arquivos `XML`, que precisa ser escrita. A pergunta que se faz é: se o framework já possui uma série de padrões predefinidos, por que repeti-los sob a forma de arquivos de configuração?

Esta redução na configuração pode parecer assustadora: estaríamos na realidade optando por uma solução menos flexível, mais "engessada"? A resposta é **não**, pois os arquivos de configuração não desaparecem. O que muda é o momento no qual sua presença se torna necessária. Em uma abordagem "tradicional" este momento surge quando iniciamos o desenvolvimento do nosso projeto, enquanto na abordagem da programação por convenção estes só se tornarão visíveis quando as convenções `propostas` pelo framework não forem adequadas ao nosso projeto.

Um exemplo comum no qual talvez precisemos tornar explícita a configuração são as classes de domínio do nosso sistema. Grails trabalha com o conceito de classes de domínio, que representam as entidades do nosso sistema e seus atributos são automaticamente persistidos em um banco de dados. Cada atributo por convenção possuirá um campo de mesmo nome na tabela onde seus dados serão armazenados. Muitas vezes já possuímos um banco de dados pronto: nesses casos, podemos substituir a convenção padrão pela nossa personalizada. Veremos mais a respeito no capítulo 5 quando trataremos do assunto em detalhes.

A estrutura de diretórios que assustou Guto é um excelente exemplo do modo como a programação por convenção pode ser aplicada. Comecemos pela descrição do diretório raiz do projeto, exposto na imagem a seguir no qual cada pasta possui uma função muito bem definida:

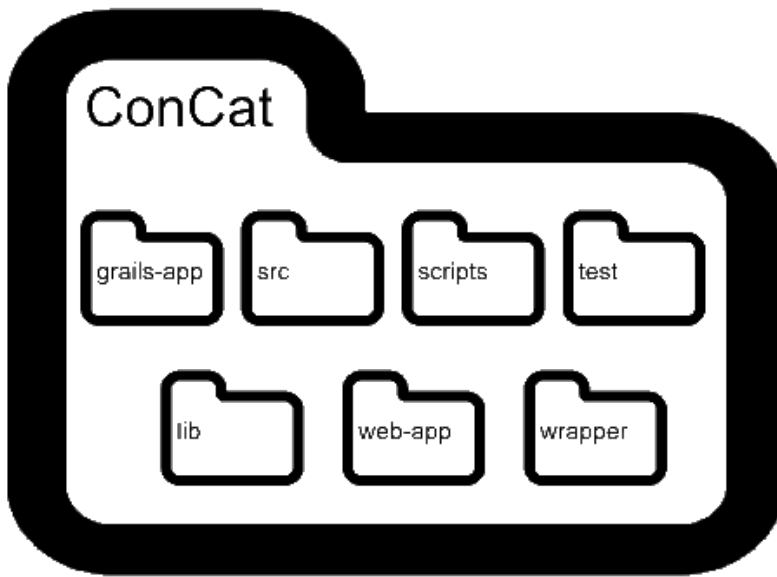



Fig. 4.7: O diretório concot

Comecemos pela pasta `src`. Muitas vezes possuímos código-fonte que gostaríamos de reaproveitar em nossos projetos. Neste caso, este diretório possui duas pastas: `groovy` e `java`, onde são armazenados os arquivos de código-fonte destas duas linguagens e que estarão acessíveis de forma transparente para o resto da aplicação.

O diretório `src/groovy` é inclusive muitas vezes usado para guardar código-fonte que não se encaixa nas convenções do Grails. O diretório `src` mostra um aspecto bastante positivo do Grails que é a facilidade com que podemos reaproveitar nosso código-fonte. Você também pode usar suas bibliotecas Java favoritas em Grails de forma transparente. Uma forma de obter este resultado é copiando os arquivos `JAR` que a compõem para a pasta `lib`. Feito isso, você poderá referenciar suas classes diretamente a partir do código-fonte do seu projeto.

Uma ferramenta importantíssima do Grails é a sua interface de linha de comando: a mesma que usamos para criar o projeto `concot`. O comando `create-app` que usamos é na realidade um script `Gant`. Gant é uma fer-

ramenta de build baseada na popular solução Ant usada por programadores Java, já faz um bom tempo. Grails nos possibilita incrementar o nosso processo de build incluindo novos scripts que são salvos nesta pasta.

Grails é um framework que nos incentiva a escrever testes para todas as funcionalidades que criemos em nosso projeto. O diretório `test` é onde armazenamos todos os nossos testes unitários, funcionais e integrados.

Como nosso projeto é uma aplicação web, precisaremos lidar com recursos estáticos, como arquivos CSS, Javascript, imagens etc. Este conteúdo é armazenado na pasta `web-app`.

Finalmente, temos aquele que é o diretório mais importante: `grails-app`. É ali que armazenaremos o código-fonte do responsável por sua lógica de negócios da nossa aplicação. Na imagem a seguir, podemos ver o seu conteúdo que é praticamente autoexplicativo.

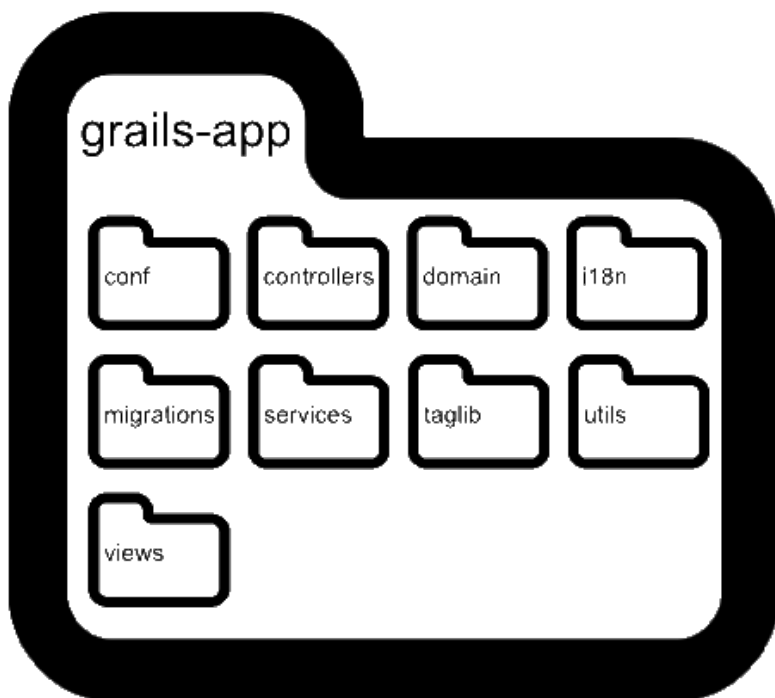



Fig. 4.8: O diretório grails-app

Estrutura de diretórios no Grails 3.0

Fig. 4.9: Compatibilidade

Há algumas pequenas mudanças na estrutura de diretórios do Grails a partir da versão 3 do framework. Falamos a respeito no último capítulo do livro 12.1.

Como dito alguns parágrafos atrás, a programação por convenção não anula os arquivos de configuração. É por essa razão que existe o diretório `conf`, onde ficam os arquivos de configuração do projeto responsáveis por definir, por exemplo, nossa conexão com o banco de dados, dependências externas, padrões de URL e diversos outros pontos da aplicação.

O padrão de projeto mais importante adotado pelo Grails é o MVC e isso é refletido na sua estrutura de diretórios. Todas as nossas classes de controle vão estar no diretório `controllers`, nossa camada de visualização no diretório `views`, nossas classes de domínio na pasta `domain` e as classes onde implementaremos nossa lógica de negócio (os serviços), na pasta `services`.

Uma aplicação Grails é internacionalizada desde o primeiro momento. Todo conteúdo textual estático do sistema fica armazenado no diretório `i18n`, que contém uma série de arquivos no formato chave-valor, sendo um por idioma.

No diretório `taglib`, armazenamos as bibliotecas de tags que vamos criar com Grails. Este é um recurso bastante interessante que nos permite reaproveitar elementos de visualização de uma forma bastante simples. No capítulo 8 veremos em detalhes como você poderá escrever as suas de uma forma bastante simples.

A versão 2.0 do Grails trouxe um recurso particularmente interessante para a equipe responsável pela gerência de configuração e DBAs: trata-se do plugin `migrations`, que nos permite acompanhar a evolução das tabelas do nosso sistema. É nesse diretório que ficam os arquivos usados por este plugin.

O último diretório é o `utils`, onde fica código-fonte que não se encaixa bem nos demais diretórios descritos nesta seção.

Como pode ser visto, a estrutura de diretórios é quase explicativa, o que minimiza a necessidade de escrevermos arquivos de configuração para definir, por exemplo, onde ficam nossos controladores, nossos serviços etc.

E o mais legal disto tudo, Guto, é que como você tem convenções bem definidas, novos programadores podem se adaptar mais facilmente ao código-fonte do nosso projeto.

Excelente Kico, isto quer dizer que podemos continuar o projeto sem você depois, certo?

É...

O que é MVC?

MVC é a sigla que usamos para denotar o padrão de projeto *Model-View-Controller*, que é o ideal quando lidamos com aplicações que possuem uma interface gráfica. Seu objetivo é simples: minimizar o acoplamento entre os componentes da aplicação, que são divididos em três categorias: o **m**odelo, a **v**isualização e o **c**ontrole.

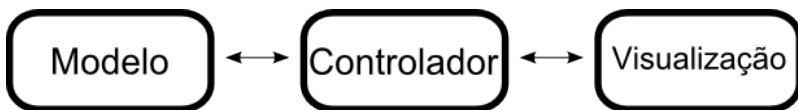



Fig. 4.10: O padrão de projeto MVC

O modelo é a camada responsável pela lógica de negócio da nossa aplicação, além de lidar com a interação do sistema com camadas inferiores, por exemplo integração com o banco de dados ou sistema de mensageria. É nesta parte do sistema onde escrevemos nossos algoritmos, como nossos dados serão persistidos, como é a integração com outros sistemas etc.

A visualização é o que o usuário de fato verá (daí o nome). Corresponde às páginas HTML ou aos formulários com os quais o usuário final do sistema irá interagir.

No meio do caminho, temos o controlador, que é o responsável por orquestrar a interação do usuário proveniente da camada de visualização e definir quais as ações que deverão ser executadas pelo modelo. Normalmente são feitas conversões de dados nessa camada. Quando acessamos uma URL pelo browser o componente ativado é o controlador.

Como o modelo não sabe da existência de um visualizador e vice-versa, consegue-se assim mínimo acoplamento entre estes componentes do sistema.

4.5 Escrevendo as classes de domínio

Agora que conhecemos a estrutura de diretórios do Grails, inicia-se o desenvolvimento do nosso projeto. Pela interface de linha de comando, vá para o diretório `concot` que criamos e, em seguida, inicie o modo interativo do Grails executando o comando `grails`. Será exposta uma saída similar à que vemos na imagem a seguir:

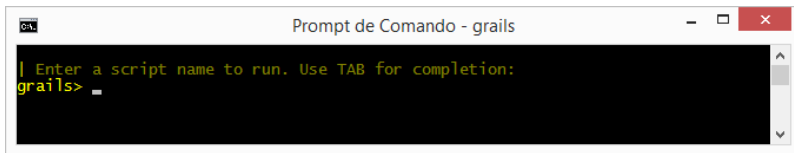



Fig. 4.11: Modo interativo ativado

Parece que não temos nada pronto, mas na realidade a aplicação que criamos já é funcional. Execute o comando `run-app` no modo interativo e, em seguida, pelo seu browser, acesse o endereço http://localhost:8080/concot. Seremos saudados por uma página de boas-vindas similar à imagem a seguir:

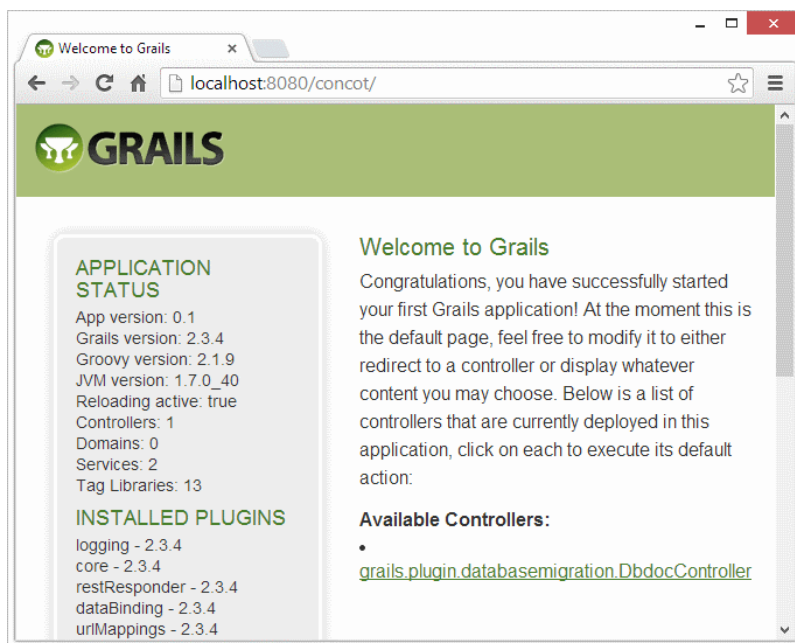



Fig. 4.12: Bem vind(a|o)!

Um framework full stack

Grails foi projetado para fornecer uma experiência de feedback imediato. A ideia é minimizar o tempo que o desenvolvedor gasta montando seu ambiente de desenvolvimento. É o que chamamos de framework *full stack*.

Muito tempo é gasto na tarefa de montar o ambiente de desenvolvimento: o programador precisa, no caso do Java, instalar um servidor de aplicações, configurar um banco de dados, instalar a linguagem de programação e integrar as diversas bibliotecas e frameworks que irão compor o seu projeto.

Um framework já vem com tudo isto pré-configurado e integrado, possibilitando ao programador gastar o seu tempo com o que realmente interessa, ou seja, na lógica de negócio do seu projeto, pois alguém já fez todo o trabalho de infraestrutura antes. Toda a arquitetura já se encontra pronta e devidamente configurada.

Apenas para mencionar, por trás dos panos uma aplicação Grails é na

realidade um projeto baseado em Spring MVC, Hibernate e diversas outras bibliotecas. Se fôssemos fazer esta integração manualmente, facilmente iríamos consumir pelo menos uma semana nesse trabalho. Com Grails, todo este processo é executado em segundos assim que terminada a execução do script `create-app`.

Dentre as configurações, encontra-se a do servidor de aplicações que, no caso do Grails, por default é o Apache Tomcat 7. Não é preciso sequer configurar o banco de dados em um primeiro momento, pois o Grails também já vem com um SGBD pré-configurado, no caso, o HSQLDB, que é 100% escrito em Java.

Criando as classes de domínio

Uma classe de domínio representa uma entidade do sistema, ou seja, um dos objetos que nosso software irá gerenciar. No caso do `ConCot`, lidamos com cinco entidades: Categoria, Item, Moeda, Cotação e Fornecedor. Mais do que representar uma entidade do sistema, o domínio também terá seus atributos persistidos em um banco de dados, garantindo que seu estado se torne permanente.

Pelo modo interativo, criamos a classe de domínio `Categoria` com o comando `create-domain-class Categoria`. Finalizada a execução do script, serão gerados dois arquivos: `grails-app/domain/concot/Categoria.groovy` e `test/unit/concot/CategoriaSpec.groovy`. O primeiro será nossa classe de domínio propriamente dita, e o segundo o esqueleto dos testes unitários que poderemos escrever para ela no futuro. Repare que o Grails o tempo inteiro nos incentiva a escrever nossos testes.

O comando `create-domain-class` será executado mais quatro vezes para gerar as classes de domínio que faltam: `Item`, `Moeda`, `Cotacao` e `Fornecedor`.

Nosso primeiro contato com as classes de domínio será com a mais simples do nosso sistema: `Categoria`, cujo código-fonte gerado pelo script `create-domain-class` podemos ver a seguir:

```
package concot
```

```
class Categoria {

    static constraints = {
    }
}
```

Vemos uma classe Groovy padrão sem muita novidade, a não ser pelo atributo estático `constraints`, que recebe como valor um bloco de código. É interessante observar que, ao executarmos o script `create-domain-class`, não fornecemos o nome do pacote, que é opcional, sendo assim foi inferido, com base nas convenções do framework, que o pacote a ser gerado consistiria no nome do projeto. Nosso próximo passo será incluir nessa classe o único atributo que possui e também adicionar algumas regras de validação. O resultado final é o código exposto a seguir:

```
package concot

class Categoria {

    String nome

    static constraints = {
        nome nullable:false, blank:false, maxSize:128, unique:true
    }
}
```

Agora ficou mais claro para que serve o bloco `constraints`. Nele, definimos as regras de validação para cada atributo da nossa classe de domínio. No caso do atributo `nome`, queremos que o valor jamais seja nulo, além de não poder inserir texto em branco e nem repetições; dessa forma, temos as regras que estão dentro do `constraints`.

Como mencionado, toda classe de domínio é persistida em um banco de dados. Por padrão, nesse banco de dados haverá uma tabela cujo nome equivalerá ao do nosso domínio em letras minúsculas. Caso a tabela não exista, será gerada pelo Hibernate, que é a ferramenta ORM usada por padrão pelo Grails.

Podemos influenciar a definição dos campos a partir do bloco `constraints`; no caso, definimos que o tamanho máximo para o campo

nome possuirá 128 caracteres usando a regra `maxSize:128`. Veremos todas estas regras, além da correlação com o banco de dados no próximo capítulo. Por enquanto, o exposto na classe `Categoria` é quase tudo o que precisamos saber.

Uma classe de domínio sozinha em nosso sistema não é algo que justifique a criação de um software. Domínios relacionam-se entre si: vejamos como ficou a classe `Item`, que se relaciona diretamente com `Categoria`.

```
package concot

class Item {

    String nome
    static belongsTo = [categoria:Categoria]

    static constraints = {
        nome nullable:false, blank:false, maxSize:128
        categoria nullable:false
    }
}
```

A novidade é o atributo estático `belongsTo`, que recebe como valor um mapa. Esse mapa possui uma única chave, `categoria`, que aponta para a classe `Categoria`. O que fizemos aqui foi um relacionamento um-para-muitos. Quando o nosso projeto for executado, a classe `Item` receberá um novo atributo, `categoria`, do tipo `Categoria`, fechando o relacionamento.

No bloco `constraints`, foi incluída uma nova linha dizendo que o atributo `categoria` não pode ser nulo. Por baixo dos panos, o que estamos fazendo é instruir o Grails para que, no momento de criação da tabela, seja também incluída uma chave estrangeira no campo `categoria` apontando para a tabela do mesmo nome.

A seguir podemos ver como ficaram as demais classes de domínio do nosso sistema, que não trazem, além dos seus atributos, novidade alguma para o leitor.

```
class Fornecedor {
```

```
    String nome

    static constraints = {
        nome nullable:false, blank:false, maxSize:128, unique:true
    }
}

class Moeda {

    String nome
    String simbolo

    static constraints = {
        nome nullable:false, blank:false, maxSize:64
        simbolo nullable:false, blank:false, maxSize:4, unique:true
    }
}

class Cotacao {

    BigDecimal valor
    Date data

    static belongsTo = [item:Item, moeda:Moeda, fornecedor:Fornecedor]


    static constraints = {
    }
}
```

Executando novamente o comando `run-app` no modo interativo, o banco de dados será gerado automaticamente e o resultado será bastante similar ao diagrama exposto a seguir:

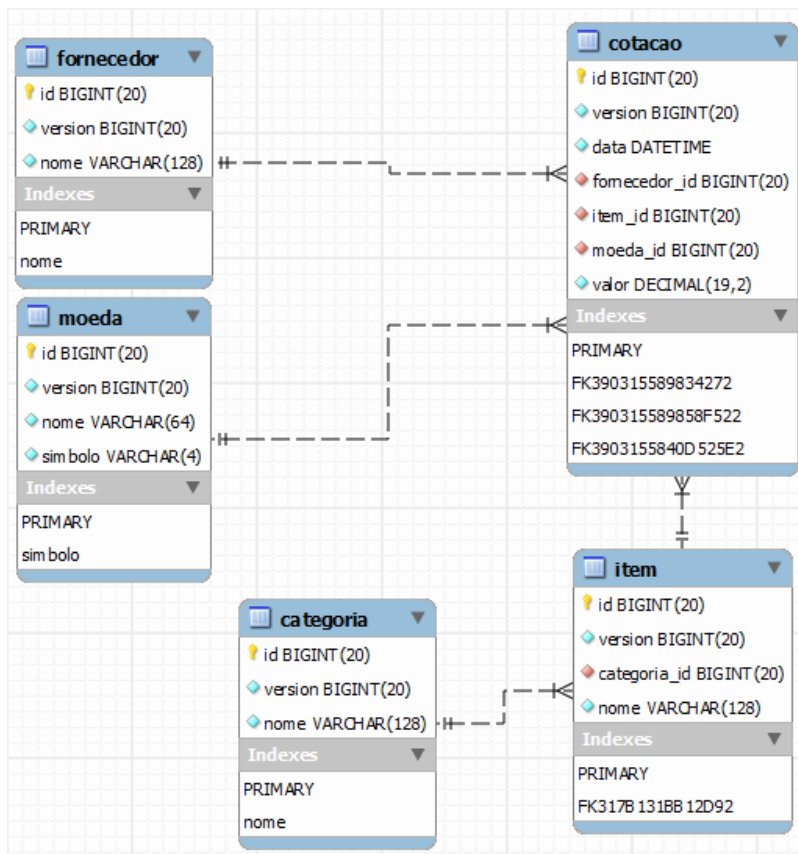



Fig. 4.13: Nosso banco de dados criado!

4.6 Dando vida ao sistema

Com nossas classes de domínio prontas, podemos dizer que temos meio caminho andado. Precisamos criar a interface com a qual nosso usuário irá interagir incluindo novos registros, gerando relatórios etc.

Vamos criar agora nosso primeiro controlador, o que usaremos para cadastrar todas as categorias do sistema. Usando o ambiente interativo, executamos o comando `create-controller Categoria` e dois arquivos são gerados:

`grails-app/controllers/concot/CategoriaController.groovy` e `test/unit/concot/CategoriaControllerSpec.groovy`

Além disso, também é criado um diretório chamado `categoria` dentro da pasta `views`. Neste momento, o que nos interessa é apenas o código-fonte do controlador, que é exposto na listagem a seguir:

```
package concot

class CategoriaController {

    def index() { }
}
```

É uma classe sem muitos atrativos e que possui um único método, chamado `index()`, que não recebe parâmetros. É interessante observar que duas convenções se manifestam na criação de um controlador:

- Toda classe de controle termina com o sufixo `Controller` em seu nome;
- Estar presente no diretório `grails-app/controllers`.

Seguindo essas simples convenções, não precisamos informar ao framework onde encontrar nossas classes de controle. Tudo fica mais simples e fácil de entender. Acessando a página inicial do projeto `ConCat`, podemos observar que algo mudou:

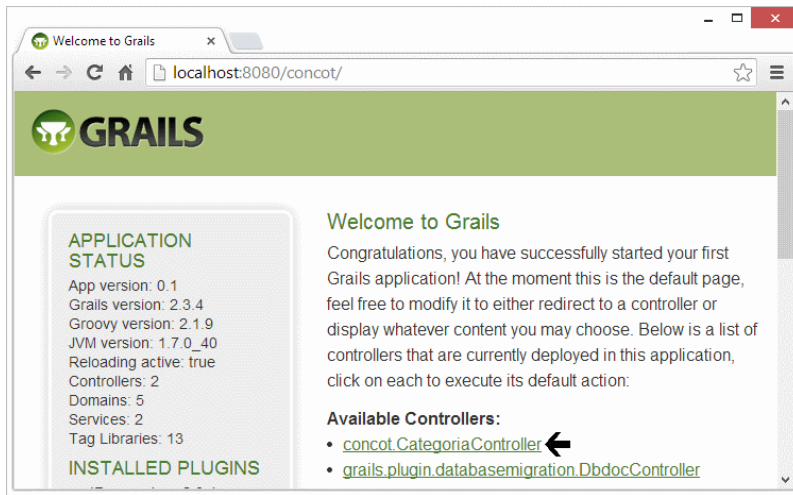



Fig. 4.14: Um novo link.

Como criamos um novo controlador, na página inicial teremos um novo link, pois, por padrão, esta tela lista todos os controladores presentes em nosso projeto.

Não clique sobre este link ainda, antes disto faremos uma pequena modificação na classe `CategoriaController` para que fique exatamente como a listagem a seguir:

```
package concot

class CategoriaController {

    static scaffold = Categoria
}
```

O novo atributo `scaffold` recebe como parâmetro a classe `Categoria`, e o resultado é simplesmente incrível: o CRUD completo da entidade é criado, sem que precisemos escrever uma única linha de HTML, Javascript ou Groovy.

O que é CRUD?

CRUD é uma sigla em inglês que denota as quatro operações mais comuns que executamos contra uma base de dados:

- `Create`: criação de registros;
- `Read`: leitura;
- `Update`: edição;
- `Delete`: exclusão.

Clicando no link que apareceu na tela inicial do projeto, observe a "mágica" que ocorreu: todo o cadastro de categorias encontra-se pronto para uso! O usuário já pode cadastrar todas as suas categorias se quiser agora, e só escrevemos uma única linha de código.

Fig. 4.15: Como mágica o cadastro está pronto

Scaffold em inglês significa "andaime". O que diversos desenvolvedores observaram é que algumas tarefas, como criação de interfaces CRUD, são bastante repetitivas e o código gerado é sempre muito semelhante. Sendo assim, pergunta-se: será que não poderíamos automatizar essa tarefa? É exatamente o que o *scaffolding* faz. No caso, automaticamente é gerada a interface gráfica **básica** de CRUD, que o desenvolvedor pode depois customizar completamente, atendendo às necessidades do seu projeto.

É importante salientar que o *scaffolding* traz apenas o básico. A ideia é fornecer apenas o que seja o mais simples e funcional possível, mas que também nos forneça flexibilidade. Veja o *scaffolding* como um pontapé inicial na escrita da sua interface gráfica. Agora, se a interface gerada lhe atender completamente, como no caso do cadastro de categorias, apenas mantenha o que o Grails gera automaticamente para você.

No entanto, é interessante observar que o "scaffold básico" do Grails é suficientemente evoluído para lidar com relacionamentos entre entidades também. Criado o controlador `ItemController`, cujo código-fonte é exposto a seguir, podemos ver como o relacionamento entre um item e sua categoria aparece automaticamente no formulário de cadastro de itens.

```
package concot

class ItemController {

    static scaffold = Item
}
```

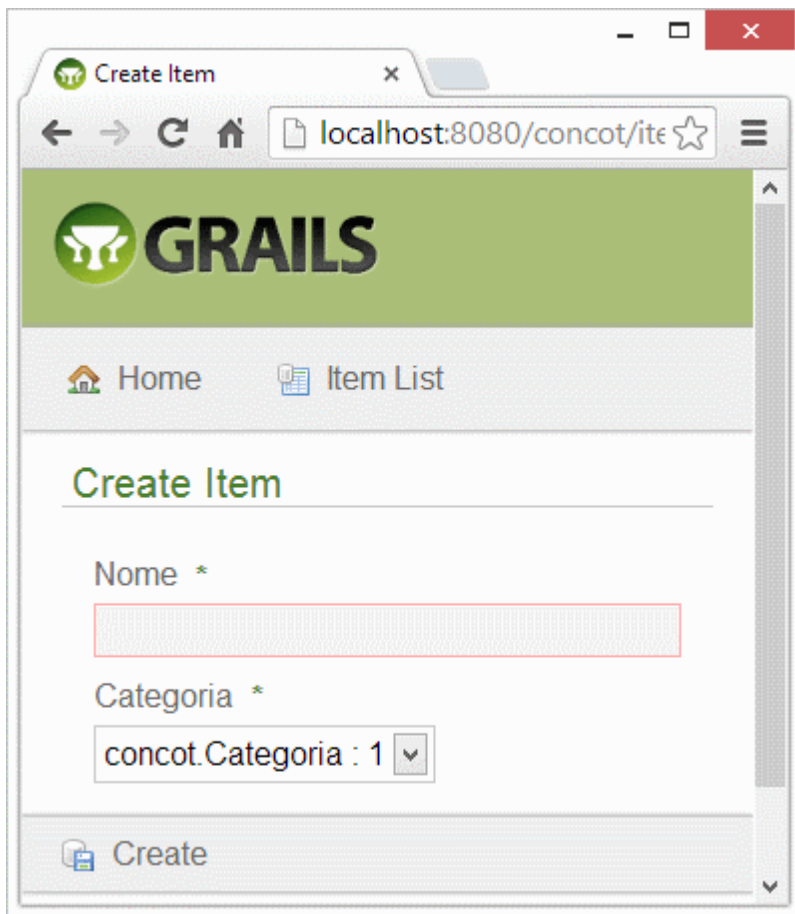



Fig. 4.16: Scaffolding de itens

Observando o *scaffolding* de itens, vemos que o modo como as categorias são expostas na caixa de seleção é bastante estranho: `concot.Categoria : 1`. O texto exposto na caixa de seleção corresponde ao resultado da função `toString()`, que é injetada em todas as classes de domínio do Grails. Sendo assim, podemos modificar a classe `Categoria` para que fique como no código a seguir, para melhorar esta representação:

```
package concot
```

```
class Categoria {

    String nome

    // Modificando o modo como a representação textual é gerada
    String toString() {
        this.nome
    }

    static constraints = {
        nome nullable:false, blank:false, maxSize:128, unique:true
    }
}
```

Revisitando a mesma página, podemos ver a sutil melhoria nos valores expostos pela caixa de seleção:

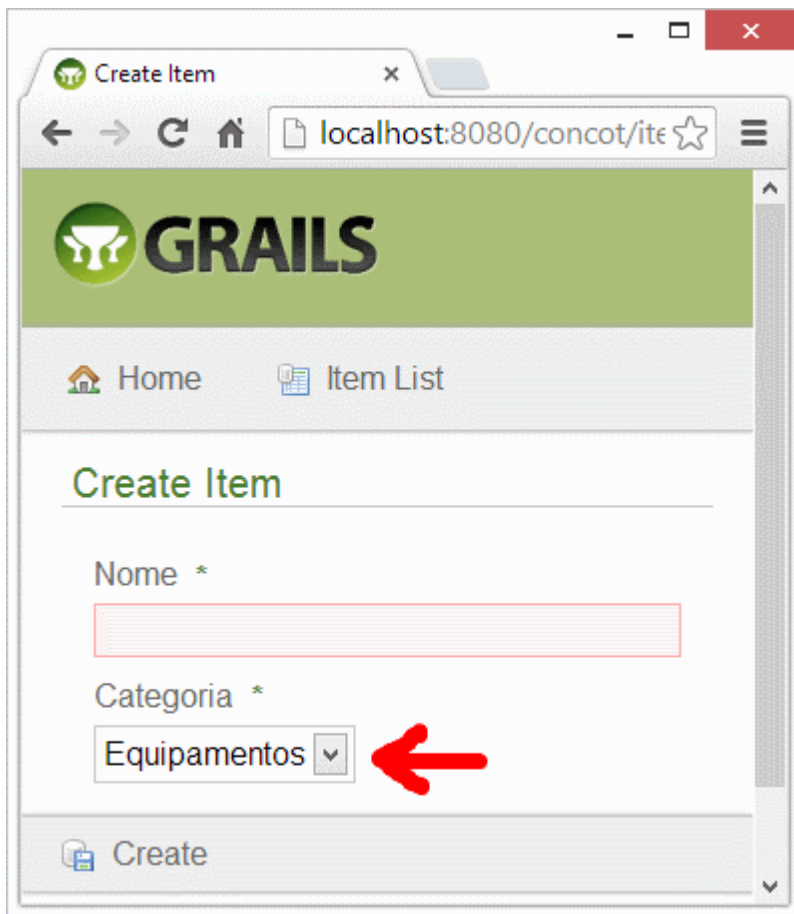



Fig. 4.17: Melhorando a representação dos itens

Para finalizar a implementação do sistema neste momento, vamos criar três controladores para as classes de domínio que ainda não possuem controladores: `Fornecedor`, `Moeda` e `Cotacao`. A seguir podemos ver o código-fonte que escrevemos:

```
package concot

class MoedaController {
```

```
    static scaffold = Moeda
}

class FornecedorController {

    static scaffold = Forecedor
}

class CotacaoController {

    static scaffold = Cotacao
}
```

Basicamente escrevendo cinco classes, cada uma com uma única linha de código, criamos toda a interface gráfica de que precisamos para iniciar o cadastro de cotações em nosso sistema *ConCot*, Guto, não é incrível?

Escrevemos tudo isto em menos de uma hora, Kico! Já posso por a mão na massa e cadastrar todas as minhas cotações?

Estamos quase lá, para finalizar o nosso trabalho de hoje, falta configurar o acesso ao banco de dados, ok?

Manda bala, Kico!

Scaffold dinâmico no Grails 3.0

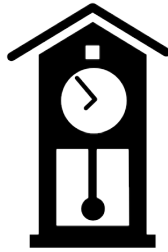



Fig. 4.18: Compatibilidade

Neste capítulo usamos o scaffold dinâmico ao declararmos o atributo estático `scaffold` em nossas classes de controle. É importante salientar que no Grails 3.0 este recurso não está presente. Para gerar o código-fonte das páginas e controladores você deverá usar o comando `generate-all`.

Mas não se assuste: o recurso deverá voltar na versão 3.1 (ou mesmo antes) do framework.

4.7 Acessando o banco de dados

Como mencionamos no início deste capítulo, Grails nos fornece um ambiente de desenvolvimento completo, incluindo o sistema gerenciador de banco de dados. A ideia é que o programador já comece o seu trabalho de forma imediata, sem precisar se preocupar em um primeiro momento em substituir componentes básicos do sistema como por exemplo o banco de dados.

Apesar de o HSQLDB ser um excelente banco de dados, ele tem um problema: **na configuração original do Grails, o HSQLDB é configurado para armazenar todos os dados apenas em memória**. A consequência é que, quando reiniciamos a aplicação ou finalizamos sua execução, os dados são

automaticamente perdidos.

É possível usar com Grails qualquer SGBD que possua um driver JDBC implementado, ou seja, praticamente todos os sistemas de bancos de dados relacionais do mercado. Só precisamos executar dois passos:

- Incluir o driver JDBC apropriado no *classpath* do sistema;
- Alterar a configuração do sistema para modificar a URL de acesso ao SGBD.

No nosso caso, vamos usar como SGBD, o MySQL, que já era inclusive usado pela *DDL* em diversos dos seus projetos. Vamos começar pelo primeiro passo, ou seja, a inclusão do driver JDBC no *classpath* da aplicação.

O driver JDBC do MySQL pode ser baixado gratuitamente em sua página oficial: http://dev.mysql.com/downloads/connector/j/. Neste site, serão expostas diversas opções, das quais a mais simples e direta consiste na distribuição em formato `zip`. Obtendo este arquivo, basta descompactar o arquivo `.jar`, `mysql-connector-java-5.x.x-bin.jar` e copiá-lo para a pasta `lib` do projeto.

O segundo passo consiste no único momento deste capítulo em que precisamos alterar um arquivo de configuração. O usuário deverá editar o arquivo `grails-app/conf/DataSource.groovy`. Não se preocupe com a extensão do arquivo. Vamos vê-lo com calma no próximo capítulo.

Neste primeiro momento tudo o que se faz necessário é alterar o arquivo `DataSources`. Primeiro, no bloco `dataSource` iremos fornecer o driver JDBC usado para estabelecer a conexão com o banco de dados, assim como as credenciais do usuário. Em seguida, no bloco `environments` iremos definir as URLs de conexão com o banco de dados para os ambientes de desenvolvimento (`development`), testes (`test`) e produção (`production`). O arquivo final será similar ao exposto a seguir:

```
dataSource {
    pooled = true
    driverClassName = "com.mysql.jdbc.Driver"
    username = "concot"
    password = "concot"
```

```
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = false
    cache.region.factory_class = 'net.sf.ehcache.hibernate.EhCacheRegionF
    dialect = 'org.hibernate.dialect.MySQL5InnoDBDialect'
}

// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "update"
            // servidor instalado localmente
            url = "jdbc:mysql://localhost:3306/concot"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            // servidor instalado localmente
            url = "jdbc:mysql://localhost:3306/concot"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            // IP do servidor
            url = "jdbc:mysql://192.168.2.20:3306/concot"
            properties {
               maxActive = -1
               minEvictableIdleTimeMillis=1800000
               timeBetweenEvictionRunsMillis=1800000
               numTestsPerEvictionRun=3
               testOnBorrow=true
               testWhileIdle=true
               testOnReturn=false
               validationQuery="SELECT 1"
               jdbcInterceptors="ConnectionState"
```

```
            }
        }
    }
}
```

Antes de executar o programa novamente, certifique-se de ter criado no MySQL um usuário e o banco de dados para o nosso sistema. Não sabe como fazer isso? Basta usar a ferramenta de linha de comando do MySQL e executar o script a seguir:

```
create database concot;
create user 'concot'@'localhost' identified by 'concot';
grant all on concot.* to 'concot'@'localhost';
```

Feito o procedimento, Guto esfrega as mãos e pergunta impacientemente a Kico:

Posso atacar agora?

Não Guto: vamos instalar a aplicação antes.

Ok...

4.8 Instalando a aplicação

O último passo para que Guto possa cadastrar suas cotações é a instalação do nosso projeto. No caso, iremos usar como servidor de aplicações o Apache Tomcat. Tudo o que precisamos fazer é, pelo modo interativo do Grails, executar o comando `war`.

O script `war` irá gerar um arquivo de deploy na pasta `target`, que será criada automaticamente no diretório raiz do projeto. Este arquivo terá o nome `concot-0.1.war`. O sufixo corresponde ao número de versão da aplicação, que é definido no arquivo `application.properties`, presente no diretório raiz.

Renomeamos o arquivo `concot-0.1.war` para `concot.war` (o que fará com que o Tomcat monte sua URL de contexto começando com "concot" e não "concot-01") e, em seguida, o copiamos para a pasta `webapps` da nossa instalação do Tomcat. Voilà: Guto já pode começar o cadastro de suas cotações.

4.9 Não se repita (*Don't Repeat Yourself* DRY)

Vamos abrir um parêntese rápido para falarmos um pouco sobre o conceito de *Don't Repeat Yourself*, que pode ser traduzido para o português como "Não se Repita" e é um dos princípios básicos por trás do Grails. Historicamente, esse princípio aparece pela primeira vez no livro *Pragmatic Programer* [17] de Andrew Hunt e David Thomas, publicado em 1999, excelente leitura por sinal.

Em um primeiro momento, parece que ele nos diz que simplesmente não devemos repetir código em nosso projeto. Na realidade, é algo mais que isso. O que de fato nos é dito é que toda informação em um sistema deve ser representada de forma única e inequívoca. E que, além disso, esta informação também pode ser a fonte para a geração de diversos outros artefatos do sistema.

Neste capítulo vimos isso de forma bem clara ao lidarmos com as classes de entidade. Cada entidade é representada centralizadamente através de uma classe presente no diretório `grails-app/domain`. Na classe de domínio estão todas as informações que a definem completamente, e ainda mais: é também a fonte que usamos para gerar dois tipos de artefato, a tabela relacionada no banco de dados e as interfaces de CRUD.

Imagine que as definições do nosso domínio estivessem em mais de um ponto como, por exemplo, em um arquivo no qual incluíssemos a definição das tabelas e também na própria classe. O código se torna mais complexo, a manutenção mais difícil e a possibilidade de erros maior. Alterando apenas a classe de domínio, podemos facilmente gerar novamente os dois artefatos que mencionei.

Mais do que isto, o princípio *DRY* também nos diz que todo processo que é repetitivo implora por automação. Mencionamos que a criação de interfaces CRUD é um processo repetitivo. Com certeza, sem o *scaffolding*, que é na prática um gerador de códigos, não conseguiríamos escrever o sistema em apenas uma hora, como fizemos neste capítulo.

4.10 Concluindo

Aqui tivemos uma visão panorâmica do modo de trabalho do Grails. Vimos como escrevendo pouquíssimo código é possível criar uma aplicação minimamente funcional. É importante avisar que esta ainda não é a aplicação completa: ainda há muito chão pela frente e nos capítulos seguintes vamos aprimorá-la até que se torne um projeto completo.

Pudemos ver na prática a aplicação dos princípios que regem o Grails, como o *Don't Repeat Yourself* e também *programação por convenções*, que minimizam drasticamente a quantidade de configuração que precisamos escrever.

Agora é hora de nos aprofundarmos nos aspectos que fazem deste um dos frameworks mais produtivos e poderosos da plataforma Java EE.

Dica: o código-fonte deste capítulo encontra-se disponível no GitHub. Basta fazer o checkout do repositório https://github.com/loboweissmann/concot

Capítulo 5

Domínio e persistência

Vamos começar nosso mergulho em Grails explorando seu componente mais complexo que é o GORM (*Grails Object Relational Mapping*), sua camada de persistência. Antes precisamos ao menos tentar responder uma pergunta: *o que é uma entidade*?

Lidando com a definição de entidade vamos apresentar o modo como as modelamos com GORM. Este será nosso ponto de partida para outros aspectos relacionados à persistência: como inserir, editar e excluir registros no banco de dados? Como escrever consultas com a ampla gama de possibilidades que o GORM nos oferece?

É um longo caminho e nele iremos topar com algumas armadilhas, mas não se assuste: elas servirão para aumentar nosso conhecimento sobre o GORM e as tecnologias que ele envolve, especialmente o Hibernate. Vale muito a pena correr alguns riscos. Que comece a dissecação do GORM: já lhe aviso, este será um capítulo árduo mas compensador!

5.1 O que é uma entidade (ou classe de domínio)?

A busca pelo conceito de entidade parte da ideia filosófica de *ente*. Ente é aquilo sobre o qual falamos a respeito [14], ou seja, é o que existe, mesmo que não seja materialmente. Por exemplo: este livro que você tem diante de si é um ente, ele existe. E sabe este conceito de entidade de que estou falando? Em si não possui qualquer materialidade, mas como falamos a seu respeito, também existe, é outro ente.

Mas apenas existir não torna algo um ente: este também deve ser diferenciável. A melhor maneira de explicar essa característica da entidade é usando um chavão: a adolescente raivosa que pega seu namorado com outra e grita aos sete ventos que *todos os homens são iguais*.

A frase *todos os homens são iguais* possui uma entidade que é o conceito de *homem*, mas como nossa personagem quer se referir ao seu namorado e não ao conceito de homem propriamente dito, ela não o reconhece como ente, pois não o diferenciou dos demais `nesta frase`.

Logo em seguida, a aflita namorada vira para sua amiga e diz: *Thomas não presta* agora sim se referindo diretamente ao seu namorado. Pelo nome, sabemos que é homem (a não ser que os pais de Thomas sejam pessoas cruéis), e também o identificamos entre todos os homens. Há um `atributo` que o diferencia, e este é seu nome. Agora sim temos a entidade *namorado (provavelmente ex agora) da adolescente raivosa*, ele se chama Thomas e é diferenciado dos demais pelo `atributo nome`. Este *grupo* do qual Thomas se diferencia os homens são o que chamamos de *classe*, *categoria* ou *conjunto*.

Todo este papo filosófico se manifesta na computação: Peter Chen [2] irá aplicá-la no seu famoso artigo em que apresenta o modelo entidade-relacionamento: *"entidade corresponde a alguma coisa do mundo real que possa ser* `identificada` *de forma direta*

. A grande diferença da *entidade* neste caso é o modo como esta se apresenta no contexto de um banco de dados e as informações que usamos para descrevê-la sob a forma de campos. Corresponde à modelagem das nossas tabelas e ao modo como garantimos a identidade de cada registro. Como? As famosas chaves primárias que estão presentes em qualquer banco de dados (relacional ou não).

No jargão Grails, ao usarmos termos como *classes de domínio* ou *domínio* nos referimos às *entidades* e ao modo como o framework as manipula através do GORM. Chegou a hora de modelarmos algumas classes de domínio.

5.2 Uma modelagem inicial

Peguemos o objetivo do *ConCot* como exemplo 4. Nosso problema é fornecer um cadastro de cotações de todos os itens que a *DDL Engenharia* usa em seus projetos. Guto e Kico, inclusive, fizeram uma modelagem inicial do sistema quando começou toda esta história sobre Grails 4.1. Toda modelagem surge de um diálogo cuja forma inicial (normalmente) é bastante simples.

Qual o objetivo do sistema?

Cadastrar cotações de itens usados nos projetos da *DDL Engenharia*.

Então nosso sistema terá `cotações`. Surge o primeiro objeto com o qual teremos de lidar: `Cotação`.

Uma cotação possui preço, data e é dada por um fornecedor, certo?

Correto, então precisamos de um segundo cadastro para incluir nossos fornecedores, que iremos relacionar com nossas cotações. Aparece a entidade `Fornecedor`.

Mas uma cotação é a cotação de alguma coisa. E `"esta coisa"` pode aparecer em diversas cotações, certo?

Como assim?

Por exemplo: imagine que precisemos cotar um caminhão de determinada marca que é oferecido por mais de um fornecedor. Esta `coisa` no caso seria nosso caminhão. Precisamos ter um cadastro deste tipo de objeto também para relacionar às nossas cotações, concorda?

Correto: é interessante termos um cadastro `destas coisas`, que chamaremos de `"itens"`. Antes que uma cotação surja, o item deve ser cadastrado. Mais uma entidade aparece: `Item`.

A cotação será feita sempre usando como moeda o Real?

Não, podemos fazer cotações em outras moedas também, como, por exemplo, o Dolar ou o Euro. Opa! Então temos de ter também um pré-cadastro de moedas, o que torna necessária a presença de uma entidade `Moeda`.

Finalmente, vamos pensar nos itens. Será que é interessante agrupá-los para gerar relatórios ou será que posso simplesmente listá-los no sistema? Por exemplo, é interessante agrupá-los em categorias como *equipamentos*, *materiais* etc.?

Sim, é bastante interessante. Assim é possível distribuir melhor as cotações na hora em que formos tomar as decisões de compra ou gerarmos relatórios. Surge o último cadastro (até este momento) de que precisamos: vamos chamá-lo de `Categoria`.

Estes objetos que mencionamos Cotação, Item, Categoria, Moeda, Fornecedor são o que chamamos de entidades ou domínio do sistema. Muitas vezes, na prática, irão corresponder a telas de cadastro, ou seja, a interface usada pelo usuário final para executarem quatro operações fundamentais em todo banco de dados: criação, edição, leitura e exclusão. Estas quatro ações possuem um nome bastante famoso entre programadores: CRUD 4.6.

É importante lembrar que nem toda entidade irá gerar uma tela de cadastro: sempre há aquelas que representam conceitos mais abstratos do sistema que não podem ser cadastradas diretamente pelo usuário final. São algo muito comum e poderemos falar a respeito mais tarde: neste momento, o que realmente nos interessa são as entidades que identificamos no sistema *ConCot* e que podemos ver representadas na imagem a seguir. As setas representam relações de dependência. Por exemplo: para incluir um registro de item no banco de dados, ele obrigatoriamente deve estar relacionado a um registro de categoria.

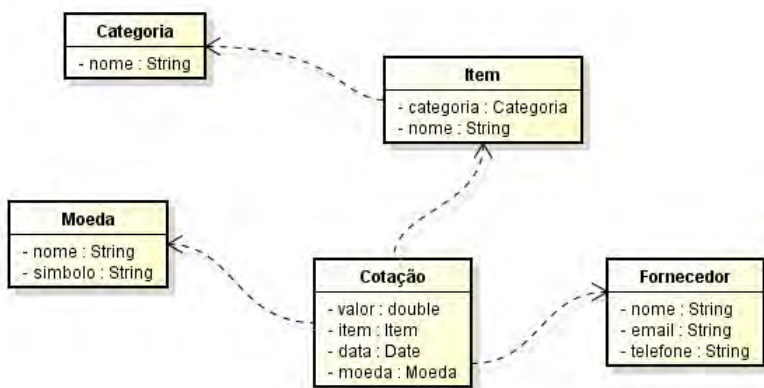



Fig. 5.1: As entidades iniciais do sistema

Voltemos ao termo *classes de domínio*. A definição dada pela equipe de desenvolvimento do Grails é: "uma classe persistente que é mapeada em relação a uma tabela presente em um banco de dados e cuja implementação se encontra no diretório `grails-app/domain`".

Ah, "entendi tudo" Kico. Persistente? Mapeada? A única parte que entendi foi a de que as classes ficam no diretório `grails-app/domain`.

Realmente é uma definição espartana, então vamos analisar alguns dos termos usados pelo pessoal do Grails. Por "persistente", entenda que o valor dos atributos dessa classe serão salvos em algum repositório de dados para posterior consulta, alteração ou exclusão. Pense em banco de dados: se existe uma classe que se encontra implementada no diretório `grails-app/domain`, esta, de acordo com as convenções do sistema deverá ser salva em algum lugar. Ser persistente equivale a "ser salvo em algum repositório de dados" como, por exemplo, uma base de dados relacional.

> Importante: se não quer que sua classe seja persistida, jamais a salve no diretório `grails-app/domain` pois TUDO o que estiver lá irá criar por padrão uma tabela no seu banco de dados.

E por falar em base de dados, voltemos à definição. A classe é *"mapeada em relação a uma tabela presente em um banco de dados*

. Que banco de dados? No caso do Grails, por padrão estamos falando de um banco de dados do tipo relacional, como MySQL, PostgreSQL, SQL Server, Firebird, Oracle e muitos outros. Cada classe de domínio possuirá neste banco de dados uma tabela cujos campos vão corresponder aos atributos presentes na classe de domínio e cujos valores vão corresponder ao estado da classe de domínio no momento em que for salva (persistida) no banco de dados.

O nome desta associação entre atributos de um objeto e campos em uma tabela se chama Mapeamento Objeto-Relacional. Em inglês, é identificado pela sigla ORM (*Object Relational Mapping*). Há diversas ferramentas na plataforma Java usadas para executar esta tarefa, por exemplo, Hibernate, Eclipse Link, Java Persistence API (JPA), Java Data Objects (JDO) e muitas outras. Neste capítulo vamos tratar daquela que o Grails adotou: o GORM.

5.3 O que é o GORM?

Esta tecnologia que possui nome engraçado é um dos principais responsáveis pelo sucesso do Grails. GORM é uma sigla, que significa *Grails Object Relational Mapping*. Trata-se de uma poderosa API que torna o lidar com tarefas relacionadas à persistência de objetos algo trivial.

Um fato marcante na história do GORM foi o lançamento da versão 2.0 do Grails. Nas versões 1.x do framework, o GORM era basicamente uma fina camada de código Groovy sobre o Hibernate, o que facilitava o uso do último ao fornecer uma interface de programação que tirasse máximo proveito das características dinâmicas do Groovy.

(Diga-se de passagem: como veremos neste capítulo, o GORM é um dos melhores exemplos de uso dos recursos dinâmicos do Groovy.)

Naquela época já havia suporte para bases de dados não relacionais (NoSQL), porém na maior parte das vezes se davam através da implementação de plugins que forneciam uma API alternativa ao GORM. O bacana desta época é que muitos programadores, ao verem as facilidades oferecidas pelo GORM, se interessaram em vê-lo como uma ferramenta que pudesse ser usada fora do Grails. Houve diversas tentativas de se isolar o GORM, mas todas até aquele momento acabaram por apresentar um "sucesso" bastante

relativo.

A partir da versão 2.0 do Grails, o GORM deixou de ser apenas esta fina camada Groovy sobre o Hibernate e passou a ser visto como uma API de persistência Groovy real. Foi inclusive criado um nome alternativo para o GORM (pouco conhecido pela maior parte dos programadores Grails brasileiros até a leitura deste livro): *Grails Datastore API*.

O objetivo da *Grails Datastore API* foi flexibilizar ao máximo os mecanismos de persistência disponibilizados aos programadores Grails. Se até a versão 2.0 havia de forma oficial apenas o Hibernate, a partir de agora é possível desenvolver sistemas em Grails que usem qualquer tipo de SGBD (Sistema Gerenciador de Banco de Dados), relacional ou não. Basta que seja escrita uma implementação da *Grails Datastore API* para o SGBD alvo.

Até a escrita deste livro, existiam as seguintes implementações oficiais do GORM: DynamoDB, Gemfire, Cassandra, Hibernate (3 e 4), JPA, MongoDB, Neo4J, Redis, Riak e SimpleDB. Neste livro iremos lidar apenas com a implementação que vêm por padrão com o Grails que é a baseada em Hibernate, mas dado que todas as demais são implementações de uma mesma API, o que for aprendido para a implementação Hibernate se aplica de forma praticamente inalterada para todas as outras.

Respondendo à pergunta que dá nome a esta seção, podemos dizer que o GORM é a API de persistência usada pelo Grails. E com um detalhe bacana: também pode ser usada hoje **independente do Grails** com sucesso.

5.4 Datasource: conectando-se ao SGBD

Feita nossa modelagem inicial, e agora que sabemos o que é o GORM, precisamos de um SGBD. Sendo assim nesta seção vamos aprender como selecionar este componente fundamental da nossa arquitetura e também como tirar máximo proveito de uma atividade `aparentemente trivial`: conectar-se ao SGBD.

Como vimos no capítulo anterior 4, Grails já vem pronto para que o desenvolvimento de aplicações possa ser iniciado da forma mais rápida possível, razão pela qual projetos Grails ao serem criados já vêm pré-configurados para se conectarem a uma base de dados gerenciada pelo SGBD H2, um sistema

gerenciador de banco de dados que pode facilmente ser embarcado em aplicações executadas na JVM por ser 100% implementado em Java.

Apesar de o H2 ser um excelente SGBD, sei muito bem que na prática é baixíssima a probabilidade de você vir a adotá-lo em seu projeto. Sendo assim, nesta seção iremos aprender com detalhes como deve ser feito o procedimento de configuração de acesso a um SGBD relacional. Neste livro usaremos o MySQL como exemplo, porém o método aqui apresentado pode ser aplicado a qualquer outro sistema gerenciador de bancos de dados que possua um driver JDBC.

Obtendo e configurando o driver JDBC

A inclusão do driver JDBC como uma das dependências do projeto é bastante simples e há duas maneiras de se executar essa tarefa. A primeira e recomendável consiste em incluí-lo no arquivo `grails-app/conf/BuildConfig.groovy`, que é usado para definir as dependências de um projeto Grails e também os parâmetros usados por todos os comandos e scripts na linha de comando do framework. Quando usamos o termo *dependência*, estamos nos referindo a toda biblioteca terceirizada (ou mesmo desenvolvida internamente pela sua equipe).

No nosso caso, a dependência será o driver JDBC (tipo 4) do MySQL. Basta alterar o bloco `dependencies` do arquivo incluindo uma linha tal como a exposta no código a seguir:

```
dependencies {
    runtime 'mysql:mysql-connector-java:5.1.27'
}
```

Esta instrução inclui uma dependência no formato Maven [**?**], claro, com um aspecto mais *groovy*. O primeiro elemento é o seu escopo, em nosso caso, `runtime`. Em seguida, é exposta sua definição. Respectivamente, separados pelo caractere dois pontos estão definidos o grupo, nome da dependência e sua versão. Quando o comando grails for executado, automaticamente o driver será baixado para o computador do usuário e estará disponível para a aplicação.

Maven e as dependências

O Apache Maven é uma ferramenta de automação de build e gestão de projetos extremamente popular na comunidade Java que fornece uma forma padronizada de compilar, empacotar, executar testes e executar projetos, que tem como ambiente de execução a JVM.

Dentre as inovações trazidas pela ferramenta, sem dúvidas a mais conhecida é o modo como gerencia as dependências de um projeto Java. Todas estas são declaradas em um arquivo padronizado (`pom.xml`) que segue uma sintaxe bastante simples definida a partir de quatro atributos:

- **group (grupo)** Identifica a organização/empresa responsável pela manutenção da biblioteca/framework.
- **name (nome)** O nome da dependência.
- **version (versão)** O número de versão da dependência.
- **scope (escopo)** Em que momento a dependência é usada no projeto. O escopo padrão de todas é o `compile`, que indica que ela será usada na compilação e execução do projeto. Um bom exemplo é o `test`, indicando que a dependência só é necessária durante a execução dos testes automatizados do projeto.

Todas as dependências gerenciadas pelo Maven são inicialmente armazenadas em **repositórios remotos** acessados pela ferramenta. Quando o Maven é executado, caso a dependência declarada para o projeto não se encontre presente no computador do programador (em seu *repositório local*), ela é baixada e, em seguida, cacheada localmente, evitando que seja necessário executar o download a partir da segunda execução do projeto.

Um ponto importante do gerenciamento de dependências do Maven é o fato de este lidar também com as dependências transitivas, ou seja, necessárias para a execução da sua dependência. Para melhor entender este conceito, pense na dependência `grails`. Como sabemos, o Grails é baseado no Spring, logo, para que possa ser executado, as classes do Spring devem

estar no classpath do seu sistema. Ao declararmos a dependência, automaticamente o Maven irá baixar para o computador do usuário todas as dependências relacionadas, poupando um enorme trabalho à equipe de desenvolvimento.

Muitas vezes, o driver JDBC não está disponível em um repositório remoto Maven, tal como ocorre com alguns SGBDs comerciais como o Oracle. Nesses casos, faz-se necessário entrar em contato com o fornecedor, normalmente através do site oficial para a obtenção do driver. No caso do MySQL, o driver (*Connector/J*) pode ser baixado em http://dev.mysql.com/downloads/connector/j/. Neste site a melhor opção é, ao selecionar a plataforma alvo, escolher a opção *Platform Independent* tal como o exposto na imagem a seguir:

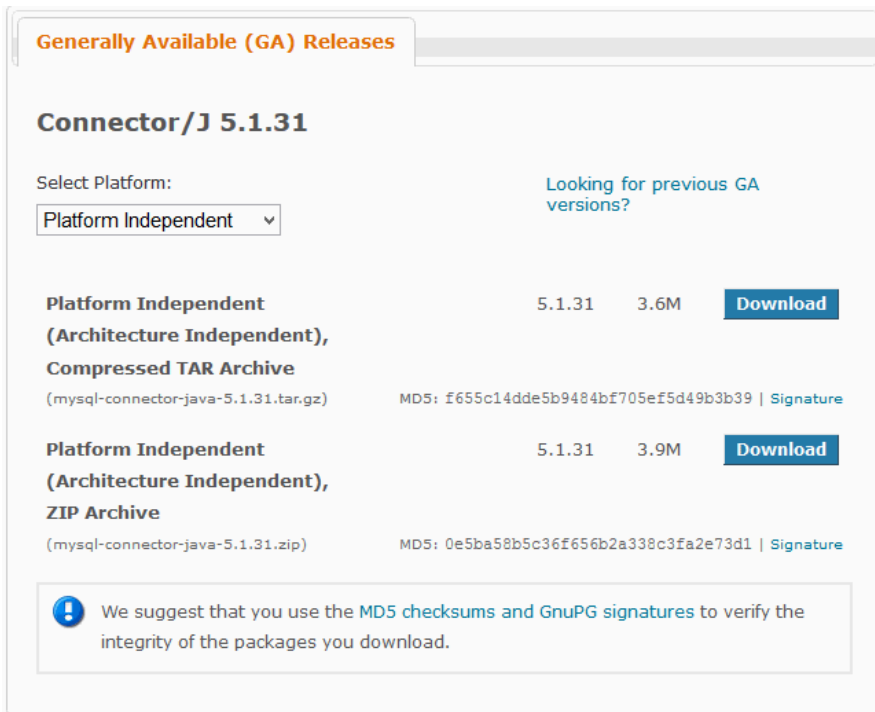



Fig. 5.2: Detalhe do site aonde podemos baixar o Connector/J

Optando pelo download do arquivo `zip` e extraindo seu conteúdo, o

programador irá encontrar uma estrutura como a seguinte:

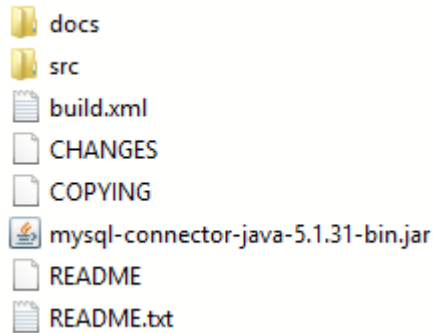



Fig. 5.3: Copiando o Connector/J para a pasta lib do projeto

Para tornar o driver disponível, basta copiar o arquivo (ou arquivos) `JAR` da distribuição do seu driver para o diretório `lib` do seu projeto Grails. Com isso, o driver já estará disponível no classpath da sua aplicação. Aliás, essa é uma excelente maneira de aproveitar qualquer código legado ou biblioteca caso esta não esteja presente em um repositório Maven: basta copiar os arquivos `JAR` para a pasta `lib`.

O que é um driver JDBC?

JDBC (*Java Database Connectivity*) é a API padrão da plataforma Java SE dedicada à interação com sistemas gerenciadores de bancos de dados relacionais (SQL).

Sozinho o JDBC não possui utilidade: navegando nas classes fornecidas pela JVM ou JDK o que o desenvolvedor encontrará será apenas uma série de classes (muitas abstratas) e interfaces. A mágica ocorre quando os fornecedores de SGBDs relacionais as usam para implementar seus drivers de acesso: os famigerados drivers JDBC.

Um driver JDBC é a implementação das interfaces e classes abstratas presentes na API JDBC do Java. Seu objetivo é possibilitar aos desenvolvedores Java interagir com o SGBD. A grande vantagem do JDBC é o fato de que,

ao programarmos contra as suas interfaces, e não diretamente contra as implementações das mesmas, passamos a ter maior mobilidade em relação ao SGBD. Precisou migrar do MySQL para o PostgreSQL? Se suas consultas forem SQL padrão basta trocar a implementação do driver.

Atualmente praticamente todos os SGBDs do mercado oferecem um driver JDBC. Para que sua aplicação interaja com o servidor de banco de dados, basta visitar o site do fabricante, buscar pelo driver e inseri-lo no classpath do seu projeto (ou, no caso do Maven, incluí-lo entre suas dependências).

Para finalizar esta história sobre drivers JDBC é importante mencionar que eles se dividem em quatro tipos:

- **Tipo 1 Ponte JDBC-ODBC**: caso não exista um driver JDBC para o seu SGBD mas exista um driver ODBC, você pode usar a versão ODBC usando este tipo de driver, que o encapsula, disponibilizando a API JDBC para o desenvolvedor.
- **Tipo 2 Driver API-Nativo**: neste caso o driver costuma formado por dois componentes. Código Java que implementa a API JDBC e código nativo que é invocado a partir do código Java.
- **Tipo 3 Driver de Protocolo de rede**: traduz chamadas JDBC para um protocolo de rede, que normalmente é o adotado pelo próprio SGBD alvo do driver.
- **Tipo 4 Driver nativo**: é o tipo ideal, 100% implementado em driver e também o mais comum. Tudo o que o desenvolvedor precisa para poder usar este driver é ter seus arquivos JAR presentes em seu classpath.

Não basta obter o driver e incluí-lo no seu classpath: é preciso também informar ao Grails que este deve ser usado. Entra em ação o arquivo `grails-app/conf/DataSource.groovy`, que é o local onde definimos todos os parâmetros relacionados à conexão com o banco de dados.

O arquivo é composto por três blocos: `dataSource`, `hibernate` (apenas quando lidamos com bases de dados relacionais) e `environments`. Neste primeiro momento iremos tratar apenas da primeira, cujo código-fonte exposto a seguir reflete a configuração do nosso projeto **ConCot** 4.

```
dataSource {
    pooled = true
    jmxExport = true
    driverClassName = "com.mysql.jdbc.Driver"
    username = "concot_user"
    password = "concot_senha"
}
```

O bloco `dataSource` normalmente é o ponto em que incluímos as configurações mais fundamentais referentes à obtenção de conexões com o banco de dados. A sintaxe adotada é bastante simples: do lado esquerdo digitamos o nome da chave de configuração, e do lado direito o valor que desejamos para a chave.

Nessa configuração definimos cinco atributos:

- **pooled**: indica se iremos usar um pool de conexões ou não. Veremos mais a respeito mais à frente nesta mesma seção.
- **jmxExport**: caso seja passado o valor `true`, indica que poderemos acompanhar o status da nossa fonte de dados a partir de um cliente JMX como, por exemplo, o JVisualVM e JConsole.
- **driverClassName**: a classe do nosso driver. Deve ser passado como valor uma string contendo o nome completo da classe do driver JDBC, ou seja, tanto o nome quanto o pacote no qual se encontra.
- **username**: o nome do usuário usado na obtenção de conexão com o banco de dados.
- **password**: a senha usada durante a obtenção de uma conexão com o banco de dados.

O atributo `driverClassName` varia de acordo com o SGBD escolhido. Para saber qual valor deve ser adotado, é necessário consultar a documentação do seu driver JDBC.

Configurando o Hibernate

Caso esteja usando um banco de dados JDBC uma seção importantíssima do arquivo `DataSource.groovy` é a `hibernate`, cujo código podemos conferir a seguir:

```
hibernate {
    dialect = 'org.hibernate.dialect.MySQL5InnoDBDialect'
    cache.use_second_level_cache = true
    cache.use_query_cache = false
}
```

Dos três atributos expostos nesse código, `dialect` é de fundamental importância, pois define qual o dialeto usado pelo Hibernate, que deve receber como valor uma string. Apesar de a maior parte dos bancos de dados relacionais adotarem o padrão SQL, sempre há variação entre um SGBD e outro resultante das características intrínsecas de cada implementação. Para resolver esse problema o Hibernate adota o conceito de dialetos.

Para cada SGBD há uma implementação da interface `org.hibernate.dialect.Dialect` que ajuda o Hibernate a, no momento em que for executar instruções SQL, escolher as variações corretas que ocorrem em cada SGBD. No caso do MySQL, há mais de um dialeto implementado, de acordo com o motor de armazenamento adotado. Como desejamos tirar proveito da integridade referencial, usaremos como dialeto o valor exposto na configuração que expusemos, que é o adotado pelo InnoDB, o motor de armazenamento do MySQL que oferece suporte a integridade referencial. Caso não esteja usando o MySQL, consulte o Apêndice 1 13 deste livro no qual encontram-se listados todos os dialetos que estão presentes na distribuição padrão do Hibernate.

Caso não esteja listada no *Apêndice 1* alguma implementação do dialeto para o seu SGBD, é grande a possibilidade de que alguém já tenha criado uma implementação para você. Neste caso, a sugestão é consultar a comunidade open source em busca de uma implementação que lhe atenda para, em seguida, incluí-la no classpath do seu projeto.

Detecção automática de dialetos

Caso a chave de configuração `dialect` não esteja presente no arquivo `DataSource.groovy`, o próprio Grails pode descobrir automaticamente qual dialeto usar.

Para tal, por debaixo dos panos, Grails na realidade registra um bean chamado `dialectDetector`, que verifica os metadados do objeto `dataSource` e, com base nestas informações, infere qual o dialeto que deve ser aplicado nas configurações.

Fig. 5.4: Atenção redobrada!

Nem sempre esta é a melhor alternativa. No caso do MySQL, desde a versão 5.5 foi definido como motor de armazenamento default o InnoDB, mas se sua versão for mais antiga, será usada a opção padrão `MyISAM`. Sendo assim, é interessante que o desenvolvedor sempre defina o valor do atributo `dialect` do bloco `dataSource`.

Os outros dois atributos definidos na chave `hibernate` são o `cache.use_second_level_cache` e `cache.use_query_cache`, que ativam ou inativam respecitvamente o cache de segundo nível e de consultas do Hibernate.

Ambientes de execução

A terceira seção do arquivo `DataSource.groovy` é a `environments`, na qual definimos quais os ambientes de execução do nosso projeto. Mas o que é um ambiente de execução? Para melhor entender este conceito, vamos pensar nos três modos como lidamos com o código-fonte do nosso sistema.

Em um primeiro modo, ocorre quando configuramos o ambiente de execução para que o sistema possa funcionar na máquina do programador. É

o que chamamos de ambiente de desenvolvimento (*development*). Normalmente neste ambiente cada programador possui instalado em seu computador o servidor de banco de dados e contra este executa o código testando as modificações conforme estas se manifestam em seu código-fonte.

Mas testar a correção do código apenas executando não é o melhor dos caminhos. Quando lidamos com Grails, o tempo inteiro o framework nos incentiva a escrever testes automatizados. Estes testes irão incluir uma série de registros em uma tabela ou outra, de vez em quando até mesmo apagar o conteúdo do banco de dados inteiro para executar uma verificação e por aí vai. Nessa situação, é interessante (praticamente obrigatório) que as bases de dados usadas para desenvolvimento e testes sejam distintas. Esse é o ambiente de testes (*test*).

Finalmente, temos o momento da entrega do sistema ao cliente que jamais poderá ser o ambiente de testes ou desenvolvimento, mas sim o de `produção` (*production*). É onde o objetivo do software se realiza: onde é implantado.

Na configuração padrão do Grails são definidos estes três ambientes: *development* (desenvolvimento), *test* (testes) e *production* (produção), o que fornece à equipe responsável pela evolução do código a configuração mínima de infraestrutura para que possa ser controlado o ciclo de vida da aplicação. No código a seguir podemos ver como estes ambientes aparecem no arquivo `DataSource.groovy` relacionadas ao projeto *ConCot* 4.

```
environments {
    development
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/concot_dev"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/concot_test"
        }
    }
    production {
```

```
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/concot"
            }
        }
    }
}
```

Na definição de cada ambiente é incluído um bloco chamado `dataSource`, que é o mesmo que vimos nesta seção quando mostramos como definir o driver JDBC. Em tempo de execução, o que ocorre é que os conteúdos das duas seções `dataSource`, a isolada no início do arquivo e a presente em cada ambiente, serão mesclados em uma só. Quando o sistema encontra-se em execução com a configuração definida aqui, o resultado na prática será uma configuração tal como a exposta a seguir:

```
development {
    dataSource {
        pooled = true
        jmxExport = true
        driverClassName = "com.mysql.jdbc.Driver"
        username = "paideia"
        password = "quentecomopaideia"
        dbCreate = "update"
        url = "jdbc:mysql://localhost/concot_dev"
    }
}
```

A fusão de configurações sempre ocorre de cima para baixo. Sendo assim, se houver uma chave definida em um ambiente que tenha o mesmo nome daquele definido no bloco `dataSource`, a de baixo irá prevalecer. É possível portanto, por exemplo, ter um ambiente usando um driver JDBC diferente daquele definido no bloco `dataSource` presente no topo do arquivo.

Assim como há um bloco `dataSource` definido para cada ambiente, é possível também ter um bloco `hibernate` customizado, como o seguinte, que ativa o cache de consultas para o ambiente de desenvolvimento.

```
development {
    hibernate {
```

```
        cache.use_query_cache = true
    }
    dataSource {
        dbCreate = "update"
        url = "jdbc:mysql://localhost/concot_dev"
    }
}
```

É possível também ter o seu próprio ambiente de execução customizado: imagine que seja necessário um ambiente chamado *beta*. Tudo o que o desenvolvedor precisa fazer é adicionar um bloco em `environments` com este nome, como exposto no código a seguir:

```
environments {
    beta {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/concot_beta"
        }
    }
}
```

A escolha do ambiente de execução no Grails, é claro, baseia-se também em convenções. Ao executar o comando `grails run-app`, o ambiente `development` será acionado, executando testes com `test-app`, o ambiente `test` e, finalmente, ao gerar o arquivo `WAR` com o comando `war`, `production` será ativado.

Para executar um ambiente customizado, basta que o programador passe o parâmetro `-Dgrails.env` na linha de comando do Grails. Para gerar um arquivo `WAR` usando as configurações do ambiente `beta` aqui exposto, portanto, bastaria executar o comando a seguir:

```
grails war -Dgrails.env=beta
```

A URL de conexão

A URL de conexão (também chamada de *"string de conexão*

), definida pelo atributo `url` do bloco `dataSource` é a responsável por especificar como deve ser feita a conexão a partir do driver JDBC. Esta aponta

para um endereço, cuja definição irá variar de acordo com o driver JDBC escolhido para o projeto.

No caso do MySQL, a URL de conexão sempre é iniciada com o prefixo `jdbc:mysql://`, seguida do nome ou IP da máquina na qual se encontra em execução o servidor de banco de dados, o caractere `/` e, finalmente, o nome do banco de dados.

Podem ser incluídos também vários atributos ao final da URL que nos permitem definir como o driver JDBC deverá se comportar. No exemplo a seguir, podemos ver a inclusão do parâmetro `autoReconnect` na string de conexão, que instrui o driver a tentar reconectar-se ao servidor em caso de erros de comunicação durante a execução de comandos SQL:

```
environments {
    beta {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/concot_beta?autoReconnect=true"
        }
    }
}
```

Muitas vezes a URL de conexão se torna enorme conforme aumenta o número de parâmetros que incluímos ao seu final. Nesses casos, há uma alternativa mais interessante para resolver este problema: você pode incluir o bloco `dbProperties` no interior de `dataSource` tal como no código a seguir:

```
environments {
    beta {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/concot_beta"
            dbProperties {
                autoReconnect=true
                jdbcCompliantTruncation=true
            }
        }
```

```
    }
}
```

A propriedades expostas nesse código são apenas algumas específicas do driver `Connector/J` do MySQL, que podem ser consultadas em http://dev.mysql.com/doc/connector-j/en/connector-j-reference-configuration-properties.html, porém o mesmo procedimento pode ser aplicado a qualquer outro driver JDBC.

Geração automática do banco de dados

Por padrão, programadores Grails não precisam se preocupar com a criação das tabelas no banco de dados. Isso porque o Hibernate o fará para nós automaticamente. É possível fazer o contrário também, ou seja, primeiro criarmos as tabelas (ou reaproveitar as existentes) e depois escrevermos as classes de domínio, mas agora vamos tratar apenas da abordagem padrão do framework.

Para tirar máximo proveito da geração automática de tabelas, é importante conhecer o parâmetro de configuração `dbCreate`, que pode receber como valor uma das opções a seguir:

- **create**: quando a aplicação é iniciada, as classes de domínio são analisadas e, em seguida, caso não exista uma tabela para alguma das classes de domínio, ela será criada. Também são avaliados os atributos das classes, o que poderá incluir novos atributos nas tabelas relacionadas.

- **create-drop**: quando a aplicação for iniciada, o comportamento definido em **create** é executado, e quando for finalizada, o banco de dados será inteiramente excluído. **PERIGO: muito cuidado com esta opção.**.

- **update**: quando a aplicação é iniciada, verifica-se a necessidade de novas tabelas serem criadas caso ainda não existam, assim como a presença ou não de campos relacionados aos atributos das classes de domínio.

- **validate**: nenhuma alteração será feita na base de dados. Apenas será verificado se todas as tabelas esperadas existem, assim como a presença dos campos. Não existindo, o desenvolvedor será informado a respeito.

Uma outra opção interessante para este atributo é simplesmente não incluí-lo na configuração do ambiente, o que irá gerar como resultado um comportamento muito parecido ao do **validate**: a diferença é que os erros poderão aparecer apenas quando o sistema tentar acessar uma tabela ou campo inexistente no banco de dados.

Em ambiente de desenvolvimento a opção mais interessante é **update**, pois o normal é que o programador tenha um conjunto de registros que usa durante o desenvolvimento das funcionalidades do sistema.

Em ambientes de produção, observa-se um comportamento similar: nas primeiras versões é muito normal a configuração original ser **update** e, posteriormente, conforme a base aumenta de tamanho e vai se tornando mais estável, **validate** passará a ser a norma.

Fig. 5.5: Atenção redobrada!

> **ARMADILHA**
>
> Evite incluir a chave `dbCreate` no bloco `dataSource` presente no início de `DataSource.groovy`: lembre-se que ela será usada como padrão nos demais ambientes caso nestes não seja definido o seu valor.

Pool de conexões

Um ponto importante a ser levado em consideração na configuração de qualquer fonte de dados é a gerência de conexões. Pode não parecer, mas uma conexão com o banco de dados é um recurso muito caro, cuja má administração irá afetar a escalabilidade, estabilidade e desempenho do sistema.

A melhor estratégia usada para administrar as conexões com o banco de dados é a adoção de um pool de conexões. Um pool nada mais é que um repositório (um cache) no qual incluímos objetos cuja criação é cara e que

possam ser reaproveitados. A ideia é bastante simples: se criar uma conexão é caro, por que jogá-la fora se podemos usá-la novamente?

Na realidade, na configuração do bloco `dataSource` que vimos até agora fizemos um pool de conexões, obra das convenções do Grails. Um pool que irá armazenar no máximo 10 conexões. Ok, você já sabe que um pool é este cache de objetos, mas como ele funciona?

Imagine um pool que contenha apenas uma conexão: quando nossa aplicação recebe uma requisição, ela usa a conexão presente neste pool e, finalizado o trabalho, o devolve ao pool para que possa ser usado por uma próxima requisição. Para um sistema com pouquíssimos acessos (digamos, um por hora), esta é uma configuração que, ao menos ingenuamente, faz algum sentido (quase nenhum na realidade).

Agora vamos imaginar que de repente outros usuários descobrem seu projeto e agora temos dois usuários que desejam acessá-la ao mesmo tempo. A primeira requisição que chegar ao sistema irá obter a conexão, deixando a requisição concorrente em estado de espera até que esta conexão esteja disponível e possa ser usada. Tendo sorte, o processamento da primeira requisição será rápido e nosso usuário irá experimentar apenas uma lentidão leve no sistema.

Para piorar um pouco mais a situação, imagine que agora três usuários tentem acessar o sistema ao mesmo tempo. Dois ficarão em estado de espera, e a situação só piora conforme nossa aplicação vai se tornando mais popular e cada vez mais pessoas querem acessá-la. A solução é aparentemente simples: basta aumentar o tamanho do nosso pool de conexões, certo? Vamos agora pensar em uma situação oposta.

A equipe de desenvolvimento da nossa aplicação espera que um número grande de usuários acesse o sistema, então configura um pool de conexões capaz de armazenar 500 conexões e aparentemente parecem ter acertado, pois na média há uns 300 usuários acessando de forma concorrente o sistema, mas observam algo interessante: há problemas de desempenho. Usando ferramentas de monitoramento não conseguem encontrar problemas em seu código Grails. O servidor não está consumindo 100% da CPU ou dos recursos computacionais.

É quando resolvem olhar para o servidor no qual se encontra em execu-

ção o SGBD, e nele, sim, o consumo de CPU está altíssimo. A razão é simples: aquela instalação não estava preparada para lidar com um número de conexões maior do que cem.

Estes dois exemplos mostram bem a complexidade envolvida na configuração do pool de conexões. Não é simplesmente definir o maior número possível, mas sim analisar os principais atores envolvidos: sua instância do SGBD e seu servidor de aplicações. Não há uma regra de ouro para a obtenção deste número. Leve isso em consideração.

As configurações do pool devem ser inseridas no bloco `properties`, presente no interior de `dataSource`, tal como no exemplo a seguir:

```
environments {
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/paideia"
            properties {
                maxIdle = 50
                maxActive = 50
                minIdle = 10
                testOnBorrow = true
            }
        }
    }
}
```

É importante mencionar que as chaves de configuração do pool podem variar de acordo com o servidor de aplicações/servlet adotado. Neste livro, será exposta a mais popular entre desenvolvedores Grails, que é a nativa do Apache Tomcat 7; no entanto os valores expostos possuem nomes que são tão comuns entre as diversas implementações que na prática acabam formando um padrão. Vamos às chaves, portanto.

As três principais chaves de configuração de um pool de conexões são aquelas que definem o seu tamanho. Em nosso caso são as seguintes:

- **maxIdle**: o número máximo de conexões que o pool deve manter prontas em modo de espera.

- **minIdle**: o número mínimo de conexões que o pool deve manter prontas em modo de espera.
- **maxActive**: o número máximo de conexões ativas que podem ser criadas ao mesmo tempo pelo pool.

Ao definirmos o número mínimo de conexões em modo de espera, quando a aplicação é iniciada, automaticamente estas conexões serão criadas e estarão prontas para uso. Esta configuração é importante pois se bem definida irá reduzir significativamente o tempo de espera que seus visitantes terão caso não haja ainda conexões disponíveis para a sua aplicação.

(Sabe aquele caso em que os usuários do seu sistema reclamam que no início do dia ele é muito lento? Não é raro a causa do problema ser a má configuração do atributo **minIdle** do pool de conexões.)

Fig. 5.6: Atenção redobrada!

O leitor deve ficar atento para uma armadilha sutil envolvendo pools de conexão: você sempre pode correr o risco de usar uma conexão inválida. Como? Simples: seu servidor SGBD irá descartar aquelas conexões que estejam inativas por muito tempo para liberar recursos. Aqui está a situação.

Imagine que sua aplicação receba diversas requisições durante o dia, mas que à noite os acessos a ela cessem. Esse é um cenário muito comum em aplicações corporativas, cujo pico de acesso ocorre durante o horário comercial. Seu pool de conexões está corretamente configurado, as conexões estão todas lá e tudo parece lindo e perfeito até que os primeiros acessos ao sistema na manhã seguinte resultam em uma série de erros de conexão com a base de dados.

Essa é uma situação bastante comum e que assombra diversos iniciantes (já vi um caso no qual a equipe de TI tinha como prática acessar o sistema pela manhã para evitar que os usuários topassem com este erro) e aqui está a razão por trás deste fenômeno e como resolver o problema.

Todo SGBD para evitar consumir toda a memória do servidor descarta aquelas conexões que encontram-se inativas por muito tempo. Ao descartar a conexão, no entanto, não avisa ao cliente que aquela conexão é inválida. E o que ocorre quando a sua aplicação tenta usar aquela conexão que foi invalidada pelo SGBD? Erro.

A solução para o problema é bastante simples: basta definir algumas chaves de configuração no seu pool, como no código a seguir:

```
environments {
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/paideia"
            properties {
                maxIdle = 50
                maxActive = 50
                minIdle = 10
                // as chaves abaixo resolvem o problema
                testOnBorrow = true
                validationQuery = "SELECT 1"
                testWhileIdle = true
                maxAge = 10 * 60000
            }
        }
    }
}
```

As quatro chaves expostas nesse código servem para garantir que toda conexão retornada pelo pool é válida. A seguir, vemos a descrição destas chaves:

- **testOnBorrow** se definido como `true`, sempre que um cliente do pool requisitar uma conexão, vai verificar se a conexão é válida. Isso é feito executando-se alguma consulta simples contra o SGBD, que naquele momento saberá que a conexão não está inativa há tanto tempo assim.
- **validationQuery** é a instrução SQL enviada ao SGBD quando verificamos se a conexão é ou não válida. Opte sempre pela instrução mais simples possível, de preferência que não requeira dado algum presente

no banco de dados. **Sempre use comandos simples e leves nesta consulta.**

- **testWhileIdle** as conexões têm sua validade verificada enquanto estão no estado `idle`. Se estiver ativada, a opção `testOnBorrow` pode ser definida com valor `false`.
- **maxAge** qual o tempo máximo de vida (em milissegundos) de uma conexão. Isso evita que conexões antigas fiquem no seu pool de conexões.

JNDI

Talvez algo tenha incomodado o leitor nesta nossa seção: as credenciais de acesso ao SGBD encontram-se expostas em todos os exemplos. Ainda pior: muitas vezes não sabemos quais os parâmetros que devemos adotar na configuração do pool de conexões. Será que é realmente seguro tornar exposta a nossa URL de acesso ao SGBD tal como temos feito? Há uma maneira melhor de fazer isto?

A resposta é sim: um dos mais poderosos recursos da plataforma Java é o JNDI (*Java Naming and Directory Interface*), que é uma API para acesso a serviços de diretórios. Um serviço de diretório fornece ao desenvolvedor uma maneira bastante prática de se disponibilizar às aplicações recursos importantes, como, por exemplo, fontes de dados (*data sources*), que é o assunto desta seção.

Todos os servidores de aplicação/servlet Java oferecem como recurso a possibilidade de prover fontes de dados via JNDI. Com isso, tudo o que o programador precisa saber para poder acessá-la será o caminho da mesma, como configurada no servidor de aplicação.

A grande vantagem desta abordagem é a melhor divisão de responsabilidades. O responsável por definir os parâmetros de pool, credenciais de acesso e tudo mais passa a ser o DBA ou aquele encarregado de administrar o SGBD, ocultando todos estes detalhes da aplicação e, ainda mais legal: simplificando ao extremo a configuração que precisamos digitar no arquivo `DataSource.groovy`. No código a seguir podemos ver como usar JNDI com Grails.

```
environments {
    production {
        dataSource {
            jndiName = "java:comp/env/seuDataSource"
        }
    }
}
```

Como pôde ser visto no código anterior, tudo o que o desenvolvedor precisará definir é o atributo `jndiName` fornecendo como valor o caminho JNDI que leva à fonte de dados. Suas credenciais de acesso e string de conexão ao SGBD não estão mais no código-fonte. E o mais interessante: agora, caso seja necessário alterar as configurações da sua fonte de dados, não será mais necessário fazer o deploy do seu arquivo `WAR` no servidor. Basta modificar a configuração no próprio servidor de aplicações e, em seguida, reiniciar a sua aplicação para que essas mudanças façam efeito.

Consulte a documentação do seu servidor de aplicações para saber como tirar proveito deste recurso.

Configuração do sistema *ConCot*

Caso queira executar o sistema *ConCot* no seu ambiente de desenvolvimento, é necessário ter instalado em seu computador o MySQL 5.5+.

Primeiro, deve ser criado um usuário chamado *concot*, com senha *concot*. No MySQL, crie este usuário com o comando a seguir:

```
create user concot@localhost identified by 'concot';
```

Devem ser criados três bancos de dados, relativos aos ambientes de produção, teste e desenvolvimento. Basta usar o script a seguir que, além de os criar, também fornece usuário total ao usuário *concot*.

```
create database concot_prod;
grant all on concot_prod.* to 'concot'@'localhost';
create database concot_dev;
grant all on concot_dev.* to 'concot'@'localhost';
create database concot_test;
grant all on concot_test.* to 'concot'@'localhost';
```

5.5 Voltando à modelagem das classes de domínio

Tudo o que poderíamos falar sobre configuração Grails neste capítulo já foi dito. Conhecemos em detalhes como nos conectamos a um SGBD e no caminho, tal como prometido, desarmamos as armadilhas que encontramos. Convenhamos, configuração não é um assunto excitante, mas escrever código sim, e a partir de agora vamos arregaçar as mangas e lidar diretamente com código Grails escrevendo nossas classes de domínio.

Vamos entender agora como o mapeamento objeto relacional do GORM funciona. Para tal, vamos voltar ao sistema *ConCot* cuja primeira implementação vimos no capítulo 4.

O domínio e sua tabela

Segundo a convenção adotada pelo Grails, as classes de domínio devem estar no diretório `grails-app/domain`. Uma vez iniciado, o framework vai varrer todas as classes presentes nesta pasta e executar as seguintes ações:

- Todas serão anotadas com `@grails.persistence.Entity` (@Entity).
- Métodos estáticos de persistência e busca serão dinamicamente inseridos.
- Dependendo da implementação adotada pelo GORM, as unidades de persistência serão automaticamente criadas no SGBD, ou seja, no caso do Hibernate, as tabelas, de acordo com a configuração definida no arquivo `DataSource.groovy` serão criadas ou modificadas.

Nesta seção iremos tratar deste terceiro passo e entender como é feito o mapeamento entre os atributos de nossas classes de domínio e suas respectivas tabelas no banco de dados.

Se está em grails-app/domain, uma tabela será criada

Uma dúvida comum entre iniciantes em Grails é a seguinte: é possível termos classes não persistentes dentro do diretório `grails-app/domain`? Resposta rápida: não.

Pela convenção, toda classe presente neste diretório irá automaticamente gerar uma tabela no banco de dados e receber novos métodos quando o Grails for iniciado. Se você quiser ter classes não persistentes, basta salvar seu código-fonte em alguma outra pasta do seu projeto, preferencialmente o `src/groovy` (ou `src/java` caso queira lidar com código Java).

Para entender o mapeamento objeto/relacional vamos primeiro analisar a classe de domínio mais simples presente no *ConCot*, que é `Categoria`, cujo código podemos ver a seguir:

```
package concot

class Categoria {

    String nome

}
```

Na nossa configuração para o ambiente de desenvolvimento definimos o atributo `create-db` como `update` usando o MySQL. Executando a aplicação e consultando o banco de dados, encontraremos uma nova tabela chamada `categoria`. O comando `show fields` do MySQL vai nos expor a sua estrutura e algumas surpresas surgirão.

```
mysql> show fields in categoria;
+---------+--------------+------+-----+---------+----------------+
| Field   | Type         | Null | Key | Default | Extra          |
+---------+--------------+------+-----+---------+----------------+
| id      | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| version | bigint(20)   | NO   |     | NULL    |                |
| nome    | varchar(255) | YES  |     | NULL    |                |
+---------+--------------+------+-----+---------+----------------+
```

E se quisermos ver qual comando SQL gerou a tabela `categoria`, basta executar a instrução `show create table categoria` no MySQL para vermos o modo como a estrutura da tabela foi gerada:

```
CREATE TABLE `categoria` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `version` bigint(20) NOT NULL,
  `nome` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `UK_prx5elpv558ah8pk8x18u56yc` (`nome`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8
```

Ok, o atributo `nome` foi declarado na classe `Categoria` e o vemos na tabela, mas este `id` e `version` não. De onde vieram? São o resultado das convenções do Grails.

Que atributos viram campos em uma tabela?

Para que os atributos em uma classe de domínio gerem campos em uma tabela relacionada, este deve suprir dois requisitos:

- Na sua declaração, o tipo deve ser bem definido, ou seja, um atributo declarado como `def nome` não irá gerar um campo, pois o GORM não saberá qual o seu tipo, porém `String nome` sim.
- Deve estar fora da lista de atributos transientes.

Caso queira definir um atributo na sua classe de domínio que não deva ser persistido na base de dados, basta declarar o atributo do tipo estático `transients` na sua classe de domínio. O exemplo a seguir nos mostra uma classe de domínio cuja senha não será persistida:

```
class Usuario {
    String senha

    static transients = ['senha']
}
```

Uma classe de domínio Grails não precisa ter declarado o atributo identificador (que irá resultar em uma chave primária). Caso não exista um atributo chamado "id" (ou configuração customizada de campos como veremos mais à frente) o GORM o criará para nós.

Reparou que não precisamos implementar nenhuma interface ou estender classe alguma ao escrevermos uma classe de domínio? Isto ocorre graças à natureza dinâmica do Groovy. No momento em que o GORM é iniciado, ele irá varrer todas as classes que se encontrem na pasta `grails-app/domain`. Para cada uma destas irá por padrão incluir os dois atributos definidos anteriormente.

A influência das restrições (constraints)

O atributo `id`, como já era de se esperar, corresponde ao atributo responsável por identificar a classe de domínio (torná-la um *ente*, tal como visto no início deste capítulo). Seu tipo irá variar de acordo com o SGBD escolhido, graças aos dialetos do Hibernate 5.4 que vimos algumas páginas atrás. No caso do Grails 2.3 e posterior, ao lidarmos com o MySQL, não é necessário definir o dialeto: por padrão este será `org.hibernate.dialect.MySQL5InnoDBDialect`. Podemos ver o reflexo disso no comando SQL que gerou nossa tabela. Repare que em seu final há a instrução `Engine=InnoDB`.

Já o campo `version` é incluído pelo Grails como estratégia de bloqueio (*lock*). Não se preocupe neste momento com este atributo, vamos tratá-lo com detalhes mais à frente neste livro.

Um ponto interessante a observar é o tamanho do campo `nome`: 255 caracteres. Este é o tamanho padrão definido pelo dialeto do Hibernate na criação destes campos. Não gostei do tamanho deste campo. Além disto, não gostaria que algum usuário cadastrasse no banco de dados uma categoria sem nome. Posso resolver essas questões aplicando algumas regras de validação (*constraints*). Basta modificar a classe para que fique como no código a seguir:

```
package concot

class Categoria {

    String nome

    static constraints = {
        nome nullable:false, blank:false, maxSize:128, unique:true
    }
}
```

O bloco estático `constraints` define as regras de validação a serem aplicadas à classe de domínio. Algumas das regras de validação vão refletir em outros dois aspectos do framework: na geração das tabelas e páginas por *scaffolding*. Não se preocupe ainda em entender como funcionam essas regras de validação. O interessante neste momento é saber os efeitos das regras de validação em nossa classe de domínio, o que podemos fazer apagando a tabela `categoria` do banco de dados e executando novamente a aplicação. O

comando `show fields in categoria` do MySQL irá nos mostrar um resultado diferente agora:

```
mysql> show fields in categoria;
+---------+--------------+------+-----+---------+----------------+
| Field   | Type         | Null | Key | Default | Extra          |
+---------+--------------+------+-----+---------+----------------+
| id      | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| version | bigint(20)   | NO   |     | NULL    |                |
| nome    | varchar(128) | NO   | UNI | NULL    |                |
+---------+--------------+------+-----+---------+----------------+
```

A regra de validação `maxSize` recebe como valor um número inteiro. Como o próprio nome já nos diz, é usada para garantir que o texto fornecido pelo usuário relativo a um campo não seja maior do que o valor definido na regra de validação. Caso seja maior, a entidade não será persistida e uma mensagem de erro será retornada ao usuário. Mais do que isto, ela também define qual o tamanho máximo de um campo em uma tabela (ou alguma outra estrutura de dados dependendo da implementação do GORM adotada pelo seu projeto).

É importante conhecer as regras de validação que influenciam a geração de tabelas. Algumas das mais populares encontram-se expostas na listagem a seguir. A lista completa você encontrará na documentação oficial do Grails.

- **nullable**: espera um valor booleano e, quando definido como `true`, irá definir o campo da tabela de tal modo que não aceite valores nulos.
- **maxSize**: espera um valor inteiro como valor, define o tamanho máximo do campo no banco de dados.
- **minSize**: é o oposto de **maxSize**; define o tamanho mínimo do campo no banco de dados.
- **unique**: espera um valor booleano e, quando definido como `true`, irá criar um índice de unicidade no campo relacionado.
- **scale**: quando aplicado a atributos do tipo `BigDecimal`, acabam definindo a escala em campos do banco de dados que lidem com valores de ponto flutuante com precisão predefinida.

5.6 Relacionando entidades

Praticamente todo sistema possui **entidades**, e não apenas **uma** entidade. E raríssimas vezes ao nos referirmos a estas classes de domínio pensamos nelas isoladamente: sempre se encontram relacionadas de alguma maneira.

Nesta seção entenderemos como lidamos com relacionamentos em Grails e de que modo você pode tirar proveito destes relacionamentos na modelagem dos seus sistemas. Mas o que queremos dizer quando usamos a palavra *relacionamento*? Responderemos a esta pergunta revisitando as classes de domínio do sistema *ConCot* 4.

Relacionamento muitos para um

Vamos começar pelo relacionamento mais simples possível: *muitos para um* (*many-to-one*), também chamado de *filho pai*. Este relacionamento nos diz algo bastante simples: para que uma entidade exista, ela obrigatoriamente deve estar relacionada a uma outra. Confuso? Vamos ilustrá-la usando o *ConCot*.

Como você deve se lembrar, o objetivo principal por trás do *ConCot* é cadastrar a cotação de diversos itens usados nos projetos da *DDL Engenharia*. Na modelagem do sistema, Kico e Guto [**?**] decidiram que não poderia haver um item que não pertencesse a uma categoria.

Com base nesta definição, o usuário do sistema ao cadastrar um item deveria, obrigatoriamente, criar primeiro sua categoria, caso ela ainda não exista no banco de dados. Um item cotado no sistema *ConCot* obrigatoriamente deve pertencer a uma categoria. Podemos representar graficamente este relacionamento usando o diagrama a seguir:

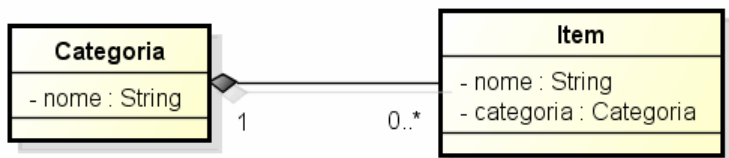



Fig. 5.7: Relacionamento muitos para um no ConCot

Mais do que isto, o relacionamento também define a essência do nosso item. Este só faz sentido enquanto pertencente a uma categoria. Talvez toda esta conversa pareça estranha neste momento, porém até o fim desta seção tudo será clarificado.

UML, agregação e composição

Neste livro, usamos a notação UML para representar o relacionamento entre nossas entidades. Caso não conheça esta notação (ou viva se esquecendo de como fazê-lo) segue aqui um resumo bem rápido.

O termo *agregação* representa relacionamentos do tipo *possui um* e o foco está no *container* da relação que pode ou não estar relacionado a outra entidade. No *ConCot* vemos a agregação ao pensarmos na categoria, que pode existir independente da presença ou não de itens.

Na imagem a seguir, vemos como representar uma agregação em UML: o losango deve ser desenhado sem estar preenchido do lado do container.

Fig. 5.8: Representando uma agregação

Ao usarmos o termo *composição*, o foco está no objeto contido, que só tem sentido (só pode existir) se relacionado com o seu container. Voltando ao exemplo do *ConCot*, o item só existe se relacionado a uma categoria.

Na imagem a seguir, vemos como representar a composição. O losango aparecerá preenchido do lado da entidade container.

Fig. 5.9: Representando uma composição

Em Grails, definir o relacionamento do tipo muitos para um é bastante simples: basta incluir um atributo na nossa classe de domínio cujo tipo seja outra classe de domínio tal como no código a seguir:

```
class Item {

    String nome
    Categoria categoria

}
```

Executando a aplicação, será criada uma tabela chamada `item` cuja estrutura podemos ver a seguir executando o comando `show fields in item` do MySQL:

```
mysql> show fields in item;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| id           | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| version      | bigint(20)   | NO   |     | NULL    |                |
| categoria_id | bigint(20)   | NO   | MUL | NULL    |                |
| nome         | varchar(128) | NO   |     | NULL    |                |
+--------------+--------------+------+-----+---------+----------------+
4 rows in set (0.03 sec)
```

O campo responsável por gerar o relacionamento entre as tabelas `item` e `categoria` se chama `categoria_id` que, como já esperávamos, possui o mesmo tipo da chave primária que vimos na tabela `categoria`. Mais do que isto, o Hibernate também criou o relacionamento entre estas tabelas. Usando uma ferramenta de modelagem como, por exemplo, o MySQL Workbench, é possível visualizar graficamente esse relacionamento.

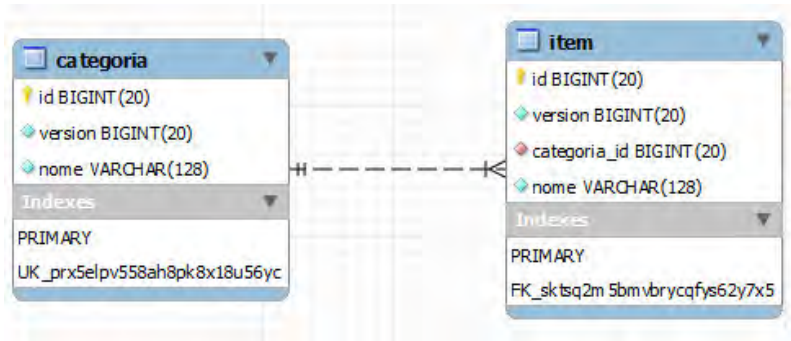



Fig. 5.10: O relacionamento entre as tabelas categoria e item

A restrição criada neste caso pelo GORM impede que sejam excluídos registros da tabela `categoria` caso exista um ou mais registros na tabela `item` que os relacione. A instrução SQL usada para gerar a tabela `item` torna isto bastante claro, como podemos conferir executando o comando `show create table item` no MySQL:

```
CREATE TABLE `item` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `version` bigint(20) NOT NULL,
  `categoria_id` bigint(20) NOT NULL,
  `nome` varchar(128) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_skts` (`categoria_id`),
  CONSTRAINT `FK_sktsq` FOREIGN KEY (`categoria_id`) REFERENCES `categoria
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Até este momento, temos um relacionamento *unidirecional*, ou seja, a classe `Item` sabe da existência da classe `Categoria`, mas o contrário não se aplica. Basta conferirmos a implementação de `Categoria`, que possui apenas um atributo para representar seu nome:

```
class Categoria {
    String nome

    static constraints = {
```

```
        nome nullable:true, blank:false, unique:true
    }
}
```

Um para muitos

Se todo item pertence a uma categoria, existirão diversos registros de itens no banco de dados referenciando o mesmo registro da tabela `categoria`. Até este momento não temos um relacionamento bidirecional entre as duas entidades (`Item` e `Categoria`). Tornaremos bidirecional o relacionamento com o atributo estático `hasMany`. Para tal, vamos modificar a nossa implementação da classe `Categoria` para que fique tal como exposto a seguir:

```
class Categoria {
    String nome

    static hasMany = [itens:Item]

    static constraints = {
        nome nullable:true, blank:false, unique:true
    }
}
```

O atributo `hasMany` quando usado deve sempre ser declarado como estático. Ele recebe como valor um mapa, no qual cada chave equivale ao nome do atributo que será criado na classe em que foi declarado pelo GORM no momento de sua inicialização e, como valor relacionado, o nome da classe contra a qual será feita a associação.

Quando uma aplicação Grails é iniciada, o GORM irá varrer todas as classes de domínio que possuam declarado o atributo `hasMany` e, para cada uma das chaves declaradas no mapa, injetará na classe um novo atributo, do tipo coleção (`java.util.Collection`) que irá referenciar todos os registros relacionados àquela entidade. Sendo assim, no caso da classe `Categoria`, após a inicialização do GORM teremos algo, em tempo de execução, similar ao código a seguir:

```
class Categoria {
    String nome
```

```
    Set<Item> itens
}
```

Mas queremos ir além: como mencionado nesta seção, o registro de um item no *ConCot* só faz sentido enquanto relacionado a uma categoria. O que realmente desejamos é instruir o GORM para que, caso uma categoria seja salva, seus itens também sejam salvos, e o mesmo comportamento também deve ser aplicado na exclusão. Entra em cena o atributo `belongsTo`.

Para tornar o nosso relacionamento bidirecional e com cascateamento completo, basta modificarmos a classe `Item` para que fique tal como no código a seguir:

```
class Item {
    String nome
    static belongsTo = [categoria:Categoria]
}
```

Assim como o atributo `hasMany`, `belongsTo` também deve ser declarado como um atributo estático e também deve receber como valor um mapa, no qual as chaves correspondam aos atributos que serão criados na classe de domínio e os valores correspondam a classes de domínio presentes em nosso projeto.

OUTRO MODO DE USAR O ATRIBUTO BELONGSTO

O atributo `belongsTo` pode receber como valor, caso sua classe de domínio possua apenas um relacionamento, apenas o nome da classe relacionada, como no exemplo a seguir:

```
class Item {
    String nome
    static belongsTo = Categoria
}
```

Neste caso será criado em tempo de execução um atributo chamado `categoria` na classe `Item`.

Modificada nossa classe `Item`, podemos escrever um código tal como o que segue:

```
import concot.*

def categoria = new Categoria(nome:"Parafuso")
categoria.addToItens(new Item(nome:"Sem porca"))
categoria.addToItens(new Item(nome:"Com porca"))
categoria.save()
```

No código do exemplo, primeiro criamos uma nova categoria com o nome `Parafuso`. Em seguida, chamamos o misterioso método `addToItens` duas vezes, instanciando objetos do tipo `Item`: um chamado `Sem porca` e outro `Com porca`. Para finalizar, chamamos o método `save` (que veremos com detalhes mais à frente) que, como o próprio nome já nos diz, persiste os dados no banco de dados.

O interessante é que ao final do script temos três registros no banco de dados: um para a tabela `categoria` e dois para a tabela `item`.

Usamos o método `addTo` para inserir elementos dentro de um dos atributos definido no mapa `hasMany` da classe container. É uma aplicação do dinamismo do Groovy, por esta razão nos referimos a este método como `addTo*`, onde `*` deve ser substituído pelo nome do atributo definido no mapa `hasMany`.

O método `addTo*` por trás dos panos executa os seguintes passos:

- Garante que existe uma coleção na classe pai relacionada ao atributo chamado. Se não existir, ele será criado.
- Na entidade dependente a ser adicionada, o atributo relacionado ao pai será preenchido.
- Inclui a instância dependente dentro da coleção relacionada na entidade pai.

Armadilha com hasMany

Fig. 5.11: Atenção redobrada!

Uma armadilha importante: evite códigos como o seguinte:

```
def categoria = new Categoria(nome:"Parafuso")
// Perigo 1
categoria.itens.add(new Item("Com porca"))
categoria.itens.add(new Item("Sem porca"))
// Perigo 2
categoria.save()
```

Há dois perigos nesse código. O primeiro é o fato de não termos certeza se o atributo `itens` da nossa instância de `Categoria` já foi instanciado (não foi). Neste caso, vamos topar com uma exceção do tipo `NullpointerException`.

O segundo perigo está no fato de que o atributo `categoria` da classe `Item` não foi definido.

5.7 Inserindo, editando e excluindo registros no banco de dados

Com nossas classes minimamente modeladas, é importante saber como inserir ou editar registros que as representem no banco de dados. No caso do Grails, isso é feito usando um único método: `save`. Podemos extrair uma série de informações a seu respeito observando o script a seguir no qual persistimos uma nova categoria.

```
def categoria = new Categoria(nome:"Cimento")
categoria.save()
```

O que estamos fazendo é bastante simples: criamos uma nova instância da classe de domínio `Categoria` que possua o nome `Cimento` e, em seguida, evocamos o método `save` na próxima linha. Com isto estamos inserindo o novo registro na base de dados que poderá ser usado por todo o restante do sistema.

Fig. 5.12: Atenção redobrada!

No entanto, imagino que o leitor goste de explorar as consequências do que acabamos de fazer. Então você executa uma consulta contra a tabela do banco de dados similar à exposta a seguir:

```
select * from categoria where nome = 'Cimento';
```

E nenhum registro aparece. Bug do GORM? Não: apenas o comportamento padrão do Hibernate. Um componente fundamental do Hibernate é a sessão. Pense nela como um cache local que armazena o estado de todos os objetos de domínio que persistimos ou obtemos a partir do banco de dados. Quando chamamos o método `save` em um objeto de domínio, por padrão apenas o marcamos para que seja persistido *eventualmente* no banco de dados.

Quão *eventualmente*? Até finalizarmos nossa sessão, sobre a qual falaremos mais à frente. Para obter máximo desempenho, o Hibernate armazena em uma fila todas as operações que deverão ser executadas contra o banco de dados e somente as envia ao SGBD quando uma das duas condições adiante ocorre:

- A fila atinge um tamanho específico.
- A sessão é fechada.

O custo de latência no envio dos comandos é significativamente reduzido graças a este comportamento do Hibernate. Em vez de termos todo o custo de enviarmos um comando, esperar o resultado do seu processamento e em seguida obter a resposta do SGBD, o fazemos em lote. Chamamos este envio em lote de instruções de *flush* da sessão.

O problema desta abordagem é que o estado do objeto fica disponível apenas no processo da JVM em que foi executada a instrução de persistência. Em um ambiente distribuído, no qual o acesso ao estado do objeto deve ser o mais atual e imediato possível, este comportamento padrão do Hibernate deve ser evitado. Como fazemos isto? Simples: basta passar um simples parâmetro ao método `save`, tal como no exemplo a seguir:

```
def categoria = new Categoria(nome:"Cimento")
categoria.save(flush:true)
```

O parâmetro *flush* com valor `true` instrui o GORM a persistir imediatamente os dados no banco de dados, evitando assim o problema mencionado. No entanto, lembre-se que haverá um custo de desempenho no caso de instruções que precisem alterar um grande número de registros.

E como edito um registro? Exatamente como faria para inserir. No exemplo a seguir alteramos o registro `categoria`.

```
def categoria = Categoria.findByNome("Cimento")
categoria.nome = "Cimento editado"
categoria.save() // agora se chamará "Cimento editado"
```

Reparou naquele método `findByNome`? Será o assunto da próxima sessão deste capítulo quando formos tratar dos métodos de pesquisa. Para finalizar, vamos reaproveitar o mesmo exemplo para excluir um registro:

```
def categoria = Categoria.findByNome("Cimento")
categoria.delete()
```

O método `delete` é usado para excluir um registro e apresenta o mesmo comportamento que o método `save`, sendo assim você pode passar o parâmetro `flush` para ele, garantindo com isto a persistência imediata na base de dados.

Validando a persistência

Classes de domínio possuem regras de validação. O que ocorre quando mandamos persistir um objeto que fira uma das restrições que definimos em nosso domínio? Vamos ver o que acontece com um experimento simples. Primeiro, leve em consideração a definição da classe `Categoria` exposta a seguir:

```
class Categoria {
    String nome

    static constraints = {
        nome blank:false, nullable:false
    }
}
```

Observe que, de acordo com as regras de validação, o nome não pode ser nulo ou uma string vazia. Agora, observe o resultado do script a seguir:

```
// criamos uma nova instância de categoria cujo atributo
// nome ainda não tenha sido definido, ou seja, será nulo
def categoria = new Categoria()
assert categoria.save() == null // passa no teste!
```

Fig. 5.13: Atenção redobrada!

O valor de retorno do método `save` é um objeto do mesmo tipo daquele que acabamos de persistir. Caso ocorra algum erro de validação, o valor retornado será nulo. O problema desta abordagem é que ela é silenciosa: nenhuma exceção será disparada por padrão e se o programador não tomar cuidado, pode ter a falsa impressão de que tudo ocorreu como esperado. Uma maneira de evitar este problema é reescrever o script tal como:

```
def categoria = new Categoria()
if (categoria.save()) {
    // tudo ok, retornou um objeto válido
} else {
    // algo deu errado. trate aqui
}
```

Uma maneira ainda mais interessante de se evitar o problema é usando o parâmetro `failOnError` com o valor `true`. Caso ocorra algum erro, uma exceção do tipo `ValidationException`. O exemplo a seguir expõe bem esta possibilidade.

```
def categoria = new Categoria()
try {
    categoria.save(failOnError:true)
} catch (ValidationException ex) {
    // trate o problema aqui
}
```

Nesse caso, é possível definir o disparo da exceção por padrão. Basta alterar o arquivo `grails-app/conf/Config.groovy` incluindo a chave `grails.gorm.failOnError`, como no exemplo:

```
grails.gorm.failOnError = true
```

É possível também definir o comportamento por classes. Para tal, basta passar como parâmetro uma lista de strings na qual cada uma represente uma das classes do seu domínio, como na listagem a seguir em que definimos o comportamento para as classes `Categoria` e `Item`.

```
grails.gorm.failOnError = ['concot.Categoria', 'concot.Item']
```

5.8 Entendendo o cascateamento

Há um "pequeno" detalhe sobre modelagem de entidades no GORM que ainda não foi mencionado: como fica a questão do cascateamento? Aliás, o que é *cascateamento* (*cascading*)?

Um conceito essencial nos bancos de dados relacionais é *integridade referencial*. Caso o leitor já esteja familiarizado com este conceito, sugiro pular para o próximo tópico deste capítulo. Valorizamos a integridade referencial quando sentirmos na pele sua falta. Imagine que o banco de dados do *ConCot* encontre-se tal como exposto na imagem a seguir:

categoria		
id	version	nome
1	0	Equipamento
2	0	Material

item			
id	version	categoria_id	nome
1	0	2	Cimento
2	0	2	Argamassa
3	0	1	Britadeira
4	**0**	**3**	**Refrigerante**

Fig. 5.14: O triste mundo sem integridade referencial

Como já sabemos, a tabela `item`, gerada a partir da definição da classe de domínio de mesmo nome possui uma chave estrangeira (`categoria_id`) que estabelece uma relação um-para-muitos com a tabela `categoria`. Observando com atenção a imagem, nota-se que o registro cujo campo `id` possui valor 4 na tabela `item` enfrenta um pequeno problema: não existe um registro equivalente na tabela `categoria` cujo identificador seja o número 4. Um item sem categoria, um registro órfão. O registro que deveria existir na tabela `categoria` é o que chamamos de *registro pai*.

Vemos duas soluções para o problema: a preguiçosa será criar um registro com o campo `id` igual a 4 na tabela `categoria`. Trata-se de uma falsa solução pois o problema não foi resolvido. Se surgiu um primeiro registro órfão, nada impede a ocorrência de um segundo, terceiro ou n-ésimo caso.

A solução real é instruir o SGBD a não permitir a criação de registros órfãos. É a *integridade referencial* que mencionei anteriormente. Para vê-la em prática, vamos executar o comando `show create table item` no MySQL para ver a estrutura da tabela `item` criada pelo GORM. O resultado será uma instrução SQL similar à exposta a seguir:

```
CREATE TABLE `item` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `version` bigint(20) NOT NULL,
```

```
  `categoria_id` bigint(20) NOT NULL,
  `nome` varchar(128) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_2323` (`categoria_id`),
  CONSTRAINT `FK_dk37` FOREIGN KEY (`categoria_id`) REFERENCES `categoria`
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Nos interessa apenas a penúltima linha da instrução na qual é criada uma chave estrangeira (`FOREIGN KEY`) para o campo `categoria_id` que referencia o campo `id` da tabela `categoria`. Se fôssemos traduzir para o português, este comando seria algo como:

"Apenas permita a inclusão de registros na tabela `item` cujo campo `categoria_id` encontre-se preenchido e referencie outro registro já existente na tabela `categoria` com o mesmo valor no campo `id`"

Mais do que evitar a inclusão de registros órfãos no banco de dados, a integridade referencial neste caso também evita a geração dos mesmos se tentarmos excluir um registro da tabela `categoria` que já esteja relacionado a pelo menos um registro na tabela `item`, o SGBD também irá bloquear a exclusão deste registro.

O que é cascateamento?

Entendido o funcionamento da integridade referencial, podemos tratar a questão do cascateamento, que é o assunto principal desta seção. Ao usarmos o termo *cascata* o leitor deve ter em mente a ideia de consequência. *Que tipo de ação ao ser aplicada a uma instância de uma classe de domínio também deve ser aplicada aos seus relacionamentos, isto é, seus filhos?*. Voltando ao GORM, vemos que há uma série de métodos de persistência que executamos contra uma classe de domínio como, *save()*, *delete()*, *merge()*. Podemos pensar na conceituação de *cascateamento* como:

Cascateamento é a regra que define se queremos ver a repetição das ações de persistência que executamos em uma entidade `pai` propagada em seus `filhos`.

O conceito fica claro com algumas perguntas que o leitor deve se fazer ao modelar suas classes de domínio. Perguntas como: *quando eu excluir, editar ou salvar minha entidade pai, devo excluir, editar ou salvar seus filhos tam-*

bém?. Como dito no parágrafo anterior, a propagação ocorre para cada um dos tipos de operação que podemos executar contra nossas classes de domínio. No entanto, neste livro (e em 90% da sua vida prática) só precisamos nos preocupar com a propagação (*propagação* é um excelente sinônimo para cascateamento) das operações `save()` e `delete()`.

Para valorizar o conceito de cascateamento, vamos proceder exatamente como fizemos ao pensar no conceito de *integridade referencial*. Vamos imaginar o mundo sem ele. Para tal, convém lembrar da definição das classes de domínio `Categoria` e `Item` expostas a seguir:

```
class Categoria {
    String nome
}

class Item {
    String nome
    static belongsTo = [categoria:Categoria]
}
```

Sem cascateamento, caso eu queira criar uma nova categoria e adicionar a ela três itens, eu deveria escrever um código como o seguinte:

```
def materiais = new Categoria(nome:"Materiais")
materiais.save()
def cimento = new Item(nome:"Cimento", categoria:materiais)
cimento.save()
def argamassa = new Item(nome:"Argamassa", categoria:materiais)
argamassa.save()
def brita = new Item(nome:"Brita", categoria:materiais)
brita.save()
```

Para cada um dos itens foi necessário chamar individualmente o método `save()`. Agora, voltemos ao mundo com cascateamento, mas antes faremos uma pequena modificação na classe `Categoria` para que tenhamos um relacionamento bidirecional:

```
class Categoria {
    String nome
```

```
    static hasMany = [itens:Item]
}
```

Ao criarmos este relacionamento bidirecional, podemos usar o método `addTo` que vimos neste capítulo e reescrever nosso script inicial de uma forma muito mais concisa:

```
new Categoria(nome:"Materiais")
    .addToItens(new Item(nome:"Cimento"))
    .addToItens(new Item(nome:"Argamassa"))
    .addToItens(new Item(nome:"Brita"))
    .save() //voilá!
```

Repare que só precisamos executar o método `save()` na entidade dominante do relacionamento que é `Categoria`. Tirando proveito do cascateamento, por trás dos panos o Hibernate executou o método `save()` primeiro em `Categoria` e, em seguida, em todos os objetos relacionados a este.

Voltemos ao exemplo anterior, e imaginemos que já temos definido em nosso banco de dados uma restrição que nos impeça de apagar um registro da tabela `categoria` caso esta já possua filhos. Sem cascateamento, para evitarmos a ocorrência de um erro precisaríamos executar um script como o exposto a seguir:

```
/* Você entenderá o que este ::findByNome:: é
   mais à frente neste capítulo.
   Por enquanto pense nele apenas como uma consulta
   que nos retorna uma instância de Categoria pelo
   nome "Materiais" */
def categoria = Categoria.findByNome("Materiais")
/* O mesmo para este método findAllByCategoria ;) */
def itens = Item.findAllByCategoria(categoria)
for (item in item) {
    item.delete()
}
categoria.delete()
```

Seria necessário apagar cada um dos registros relacionados e só ao final apagar o registro da categoria cujo nome é `Materiais`. Como é o mundo com cascateamento? Exatamente como no script a seguir:

```
def categoria = Categoria.findByNome("Materiais")
categoria.delete() //todos os filhos serão apagados
```

Pelo cascateamento, é como se o método `delete()` fosse chamado para todos os filhos da instância de `Categoria` que encontramos. "Como se", pois na prática o Hibernate otimizará este processo para evitar o consumo excessivo e desnecessário dos seus recursos computacionais.

`Isto` é o que chamamos de *propagação* ou *cascateamento* de operações de persistência. Reduzimos a quantidade de código que precisaríamos digitar, obtivemos um código mais limpo e com uma possibilidade bastante menor de ter algum bug e uma **maior garantia de que manteremos a integridade referencial do nosso banco de dados**.

Ao leitor mais curioso que queira saber mais detalhes sobre como o cascateamento realmente é implementado, recomendo a leitura da seção *Transitive Persistence* da documentação oficial do Hibernate [16].

Cascateamento no GORM

Com os conceitos bem colocados e entendida a razão pela qual existe o cascateamento podemos agora ver como o GORM lida com este conceito. A chave para tirar máximo proveito deste recurso é entender que o atributo estático `belongsTo` é o responsável pela definição dos cascateamentos no GORM.

Quando em `Item` adicionamos a definição `belongsTo = [categoria:Categoria]` estamos informando ao GORM qual classe `domina` aquele relacionamento, neste caso, `Categoria`. Lembre-se: **o cascateamento sempre é direcionado do pai para seus filhos**.

Tanto faz se estamos lidando com um relacionamento do tipo *um-para-um*, *um-para-muitos* ou *muitos-para-muitos*. Sempre que estiver presente o atributo `belongsTo` na definição de um relacionamento, as operações `save()` e `delete()` serão propagadas da entidade pai em direção aos seus filhos.

E quando não adicionamos o atributo `belongsTo` na definição de um relacionamento? Nesse caso, nenhum cascateamento irá ocorrer e terminaremos na situação exposta no tópico anterior desta seção quando descrevemos o mundo "sem cascateamento". Há uma exceção, no entanto, que ocorre

quando usamos apenas o atributo hasMany em nossas classes de domínio. Nesse caso, a entidade que contém o atributo é tida como pai e a operação save() será propagada às entidades filhas relacionadas.

Para finalizar esta seção, é importante que você conheça as convenções adotadas pelo Grails com relação às configurações de cascateamento que irão variar de acordo com o modo como modelamos os relacionamentos entre nossas entidades. Vale lembrar que podemos customizar estes comportamentos caso seja de nosso interesse. Para entender posteriormente como isto deve ser feito, recomendo que o leitor leia o capítulo [**?**], deste livro no qual descrevemos como customizar o mapeamento objeto relacional do GORM.

O esquema exposto a seguir é inspirado na documentação oficial do Grails [5] e expõe bem como as configurações de cascateamento variam de acordo com a modelagem dos nossos relacionamentos.

Relacionamento bidirecional um-para-muitos usando belongsTo

Usado quando definimos relacionamentos similares ao exposto a seguir:

```
class Pai {
    static hasMany = [filhos:Filho]
}

class Filho {
    static belongsTo = [pai:Pai]
}
```

Na classe Pai o cascateamento encontra-se configurado por padrão como ALL, ou seja, toda operação de persistência automaticamente será cascateada nos filhos também. Já do lado filho o cascateamento encontra-se definido como NONE, ou seja, operações de persistência nos filhos não serão replicadas nas entidades pai.

Relacionamento unidirecional um-para-muitos usando apenas hasMany

```
class Pai {
    static hasMany = [filhos:Filho]
}

class Filho {
```

```
}
```

O cascateamento estará ativado para as ações de inserção (SAVE) e edição (UPDATE) no sentido Pai->Filho.

Relacionamento bidirecional um-para-muitos sem usar belongsTo

```
class Pai {
    static hasMany = [filhos:Filho]
}

class Filho {
    Pai pai
}
```

Do lado pai, a estratégia de cascateamento será para as operações de inserção (SAVE) e edição (UPDATE). Já do lado filho as operações de persistência não irão cascatear nas entidades pai (NONE).

Relacionamento unidirecional um-para-um sem belongsTo

```
class Pai {


}
class Filho {
    static belongsTo = [pai:Pai]
}
```

Do lado pai o cascateamento será do tipo ALL, ou seja, abarcará todas as operações de persistência enquanto do lado filho não haverá cascateamento (NONE) algum chegando às entidades pai declaradas.

5.9 Customizando o mapeamento

Muitas vezes, as configurações padrão de mapeamento adotadas pelo GORM não se adequam às políticas de gestão de banco de dados do seu cliente ou empresa. Nesses casos, podemos alterar seu comportamento para que possa atender a estes requisitos de uma forma bastante simples usando o bloco

`mapping`, que nos permite customizar ainda mais o modo como o mapeamento objeto relacional do GORM é feito.

Outra razão pela qual é interessante customizar o mapeamento ocorre quando você já possui um banco de dados pronto e quer apenas usar suas tabelas já criadas em sua aplicação, o que também é bastante comum.

No exemplo a seguir podemos ver como declarar o bloco `mapping` em nossas classes de domínio:

```
class Cotacao {
    static mapping = {

    }
}
```

Alterando o nome da tabela (e seu esquema e catálogo quando necessário)

Para alterar o nome da tabela que deverá receber os dados da nossa entidade, basta incluir a instrução `table` tal como no exemplo a seguir na qual iremos mudar o nome da tabela padrão incluindo o prefixo `DDL`.

```
class Cotacao {
    static mapping = {
        /*
            Ao invés do padrão "cotacao",
            o nome da tabela aonde armazenaremos
            nossos dados será agora "DDL_cotacao"
        */
        table "DDL_cotacao"
    }
}
```

Em situações nas quais você precise customizar o esquema ou catálogo no qual a tabela se encontra, dois novos parâmetros podem ser passados: `schema` e `catalog`, como a seguir:

```
class Cotacao {
    static mapping = {
```

```
        table name:"DDL_cotacao", schema:"dbo", catalog:"cotacoes"
    }
}
```

Observe que se formos passar mais de um parâmetro para a instrução `table` precisamos nomear explicitamente todos os que estiverem presentes. Apenas quando alteramos apenas o nome da tabela passamos um único valor.

O leitor deve ter observado que é incluído um campo chamado `version` em todas as nossas tabelas. Ele é usado pela política de bloqueio otimista (*optimistic locking*) adotada por padrão pelo GORM.

Bloqueio otimista é um recurso disponibilizado pelo Hibernate no qual é criado um campo de versionamento em que, a cada atualização (*update*) de um registro, o valor do mesmo é incrementado. O objetivo deste recurso é tratar o acesso concorrente a um mesmo registro no banco de dados. Para entender seu funcionamento, imagine dois usuários simultaneamente editando a mesma cotação no *ConCot*.

```
//Usuario 1 pega a cotacao de id 34
//Neste momento, o valor do campo version é 1
def cotacao1 = Cotacao.get(34)
cotacao1.valor = 56


//Paralelamente, o usuário 2 pega a mesma cotaçaõ
def cotacao2 = Cotacao.get(34)
//O valor do campo version ainda é 1
cotacao2.valor = 4
// É persistida a cotação e automaticamente o
// campo version será incrementado, passando a
// ter o valor 2
cotacao2.save(flush:true)

// Usuário 1 demora um pouco mais em seu processamento
// e irá atualizar o registro
cotacao1.save(flush:true)
```

Por trás dos panos, no momento em que o primeiro usuário tentar persistir os dados da cotação, o Hibernate irá verifi-

car se o valor do campo `version` na tabela `cotacao` bate com o presente em memória. Se for diferente, uma exceção do tipo `org.hibernate.StaleObjectStateException` será disparada e, caso esteja executando o código em uma transação, o rollback será executado.

Claro, você pode desabilitar o lock otimista em sua entidade. Para tal, basta passar o valor `false` para a instrução `version` no bloco `mapping` tal como no exemplo a seguir:

```
class Cotacao {
    static mapping {
        version false
    }
}
```

Talvez o nome do campo `version` conflite com sua tabela preexistente no banco de dados. Neste caso, basta passar como valor o nome do campo tal como no exemplo a seguir:

```
class Cotacao {
    static mapping {
        /*
            Ao invés de "version", "bloqueio",
            e o bloqueio otimista é mantido
        */
        version "bloqueio"
    }
}
```

Customizando o identificador

Por padrão para a maior parte dos SGBDs do mercado (ao menos todos os que conheço e já trabalhei diretamente) GORM irá usar a estratégia do valor autoincremental para alimentar a chave primária da tabela relacionada à sua classe de domínio.

Nem sempre esta é a melhor estratégia. Talvez você queira que o valor da sua chave primária use uma estratégia como UUID [26] que armazenará seus

valores em uma String em vez de um valor numérico, ou mesmo deseje usar chaves compostas para definir o identificador da sua tabela.

Primeiro, vamos ver como modificar o tipo e estratégia de geração de dados para a chave primária. Quer que seu campo `id` contenha texto e use a estratégia UUID [26]? O primeiro passo é declará-lo diretamente em sua classe de domínio como `String` e, em seguida, usar a instrução `id`, como no exemplo a seguir:

```
class Cotacao {
    String id

    static mapping = {
        /*
            Atributo generator me diz qual estratégia de
            geração adotar
            E column me permite mudar inclusive o nome do
            campo id na tabela!
        */
        id id generator:'uuid', column:'identificador'
    }
}
```

Há diversas estratégias de geração de identificadores diferentes. Todas estão diretamente relacionadas ao Hibernate e são demasiadamente variadas para poder ser tratadas aqui, mas você pode consultar sua documentação oficial para conhecê-las melhor [7].

Você também pode definir que a identificação da sua entidade é composta por mais de um atributo. Para isso, basta usar o parâmetro `composite`, que recebe como valor uma lista contendo os campos que irão gerar sua chave composta:

```
class Cotacao {
    static mapping = {
        /*
            Id gerado pelos atributos item e
            fabricante.
        */
        id composite:['item','fabricante']
```

```
    }
}
```

Alterando a ordem padrão nas consultas

Você também pode customizar a ordem na qual os elementos de uma consulta às suas classes de domínio serão retornados. Para tal, basta usar as instruções `sort` e `order`, como no exemplo a seguir no qual definimos que todas as cotações virão ordenadas por valor em ordem decrescente:

```
class Cotacao {
    static mapping = {
        sort: 'valor'
        order: 'desc' // 'asc' para ordem crescente, que é o padrão
    }
}
```

Claro, isso só será aplicado quando você não definir a ordenação em suas consultas. Veja a seguir, usando um finder dinâmico:

```
//Valores retornarão ordenados por valor
//em ordem decrescente
Cotacao.list()
```

Customizando as demais colunas

Claro que a customização não para por aqui, você também pode alterar o modo como são mapeados os demais atributos da sua classe de domínio. Para tal, basta incluir o nome do atributo no bloco mapping e, em seguida, aplicar alguns parâmetros como no seguinte exemplo, em que customizamos fortemente a entidade `Usuario`:

```
class Usuario {
    String id
    String login
    String senha

    static mapping = {
        id generator:'uuid'
```

```
        table 'DDL_usuarios'
        version false
        login column:'username', sqlType:'char', updatable:false
        senha column:'password', sqlType:'char', updatable:false
    }
}
```

Há uma lista significativa de parâmetros que podem ser usados no mapeamento dos seus atributos. Dentre eles, os mais comuns são listados a seguir:

- `column`: recebe uma string como valor representando o nome da coluna a ser usada na tabela.
- `sqlType`: você pode customizar qual tipo SQL a ser usado na tabela. Neste caso, você deve consultar a documentação do seu SGBD para descobrir quais os valores disponíveis.
- `updatable`: define se o campo é atualizável ou não, permitindo que seu valor seja definido apenas no momento de inserção. Valor padrão: `true`.
- `insertable`: define que o valor do campo pode ser definido apenas por atualização, e não na inserção. Valor padrão: `true`.
- `length`: qual o tamanho máximo do campo na tabela.

Dica: mapeie views!

Muitas vezes encontramos situações em bases de dados legadas nas quais precisamos de uma entidade mas esta não é facilmente mapeada para uma tabela específica, mas sim um conjunto de tabelas. Nesses casos, uma opção interessante é usar o mapeamento customizado contra uma view. Você perderá a capacidade de alterar os dados, mas pelo menos terá uma opção interessante para visualização e consultas!

Mapeando views, você pode, inclusive, tirar proveito dos poderosos recursos de consulta do Grails que veremos no próximo capítulo para criar aplicações que facilitem aos seus usuários extrair importantes informações de suas bases de dados legadas.

5.10 Lidando com herança

Antes de finalizar este capítulo é preciso falar do "elefante branco" que está entre nós e do qual até agora não falamos a respeito: como lidar com herança? Afinal de contas, estamos lidando com uma ferramenta que faz o mapeamento **objeto** relacional. :)

Infelizmente, no *ConCot* não há um exemplo de herança, sendo assim por que não usar um que é bastante popular? Cadastro de "pessoas". Na legislação brasileira há dois tipos de "pessoa": física e jurídica. O primeiro tipo corresponde a indivíduos como eu e você, e a segunda a empresas.

Uma modelagem simples desta situação pode ser vista no diagrama a seguir:

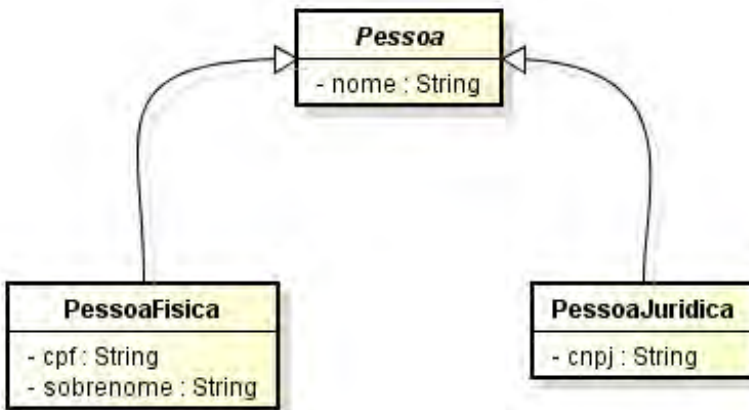



Fig. 5.15: Nossas pessoas

E a implementação inicial das nossas classes de domínio é bastante modesta, como pode ser vista na sequência:

```
abstract class Pessoa {
    String nome
}

class PessoaFisica extends Pessoa {
    String sobrenome
    String cpf
}

class PessoaJuridica extends Pessoa {
    String cnpj
}
```

Claro, você está curioso e gostaria de ver como as tabelas ficariam no banco de dados, então irá executar o comando `run-app` para ver o resultado, certo? Por padrão, sem qualquer configuração de mapeamento customizado, o que teremos é a geração de uma única tabela chamada `pessoa`, cuja estrutura podemos ver a seguir:

Field	Type	Null	Key	Default	Extra
id	bigint(20)	NO	PRI	NULL	auto_increment
version	bigint(20)	NO		NULL	
nome	varchar(255)	NO		NULL	
class	varchar(255)	NO		NULL	
cnpj	varchar(255)	YES		NULL	
cpf	varchar(255)	YES		NULL	
sobrenome	varchar(255)	YES		NULL	

O Hibernate trabalha essencialmente com duas estratégias ao mapear herança: uma tabela por hierarquia, que é o padrão (*table per hierarchy*) ou uma tabela por classe. Como pôde ser visto na imagem, todos os atributos que definimos nas classes `PessoaJuridica` e `PessoaFisica` encontram-se armazenados na mesma tabela.

Observe que um novo campo foi adicionado, `class`, que será usado pelo GORM para fazer a busca por pessoas de acordo com o tipo da classe, fazendo a diferenciação entre os tipos.

O mais importante a ser observado na estratégia "uma tabela por hierarquia" é que atributos definidos nas classes derivadas serão mapeados para campos que obrigatoriamente deverão aceitar `null` como valor. Entender este comportamento é fácil. Imagine que o campo `cnpj` não aceite `null` como valor. Na primeira inserção de uma pessoa física, o SGBD lhe retornaria um erro.

Para desabilitar o comportamento de "uma tabela por hierarquia" basta adicionar um único comando no bloco `mapping` da classe que se encontra na raíz da hierarquia de herança (`Pessoa`) tal como no exemplo:

```
class Pessoa {
    String nome

    static mapping = {
        tablePerHierarchy false
    }
}
```

O resultado é que agora teremos três tabelas: `pessoa`, `pessoa_fisica` e `pessoa_juridica`.

```
mysql> show fields in pessoa;
+---------+--------------+------+-----+---------+----------------+
| Field   | Type         | Null | Key | Default | Extra          |
+---------+--------------+------+-----+---------+----------------+
| id      | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| version | bigint(20)   | NO   |     | NULL    |                |
| nome    | varchar(255) | NO   |     | NULL    |                |
+---------+--------------+------+-----+---------+----------------+
3 rows in set (0.01 sec)

mysql> show fields in pessoa_fisica;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| id        | bigint(20)   | NO   | PRI | NULL    |       |
| cpf       | varchar(255) | NO   |     | NULL    |       |
| sobrenome | varchar(255) | NO   |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)

mysql> show fields in pessoa_juridica;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | bigint(20)   | NO   | PRI | NULL    |       |
| cnpj  | varchar(255) | NO   |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.02 sec)
```

Fig. 5.17: Comportamento da herança nas tabelas

Serão criadas chaves estrangeiras relacionando o campo `id` das tabelas `pessoa_fisica` e `pessoa_juridica` com o campo `id` da tabela `pessoa`.

Bastante simples, não é mesmo?

5.11 Concluindo

Neste capítulo foi exposto com detalhes como funciona o mapeamento de classes de domínio com Grails. Aprendemos como modelar os relacionamentos entre nossas classes de domínio, como nossos atributos se transformam em campos nas tabelas e ainda entendemos com detalhes o processo de persistência dos dados.

Esta é a metade do caminho. Agora iremos entender como tirar proveito das diferentes tecnologias de pesquisa e obtenção de dados que o Grails nos fornece. E claro: algumas armadilhas aparecerão diante de nós, mas não se preocupe pois todas serão desarmadas e você ainda aprenderá a tirar proveito delas.

Capítulo 6

Buscas

No capítulo 5 aprendemos com detalhes como modelar nossas classes de domínio em Grails e também pudemos nos aprofundar no modo como é feito o processo de persistência de dados em um banco de dados relacional. Neste capítulo (que será bem mais dinâmico que o anterior) vamos agora entender como encontrar informações em nosso banco de dados.

Grails nos fornece uma ampla gama de tecnologias voltadas à obtenção de dados:

- Finders dinâmicos
- Criterias
- Buscas where
- HQL (Hibernate Query Language)

Entenderemos aqui qual o ambiente ideal para cada uma destas alternativas e ainda exporemos algumas dicas que tornarão sua vida com Grails muito mais produtiva. Claro, usaremos como referência o nosso sistema *ConCot* que desenvolvemos até aqui.

E, sim, durante o percurso iremos desarmar algumas armadilhas que costumam pregar sustos nos iniciantes e dificilmente são descobertas.

6.1 Grails Console: nosso melhor amigo

Lembra do *Groovy Console* 2.4? A equipe responsável pelo desenvolvimento do Grails reaproveitou a ideia no framework, o que nos possibilita experimentar instruções contra as classes da nossa aplicação e todos os recursos do Grails exatamente como fizemos com código Groovy.

O console é portanto uma ferramenta **essencial** para o aprendizado do framework, especialmente as possibilidades de consulta do GORM de que iremos tratar neste capítulo. Para iniciar o console basta executar o comando `console` na interface de linha de comando do Grails. Na imagem a seguir podemos ver a execução de uma consulta simples usando finders dinâmicos (nosso próximo assunto) no console:

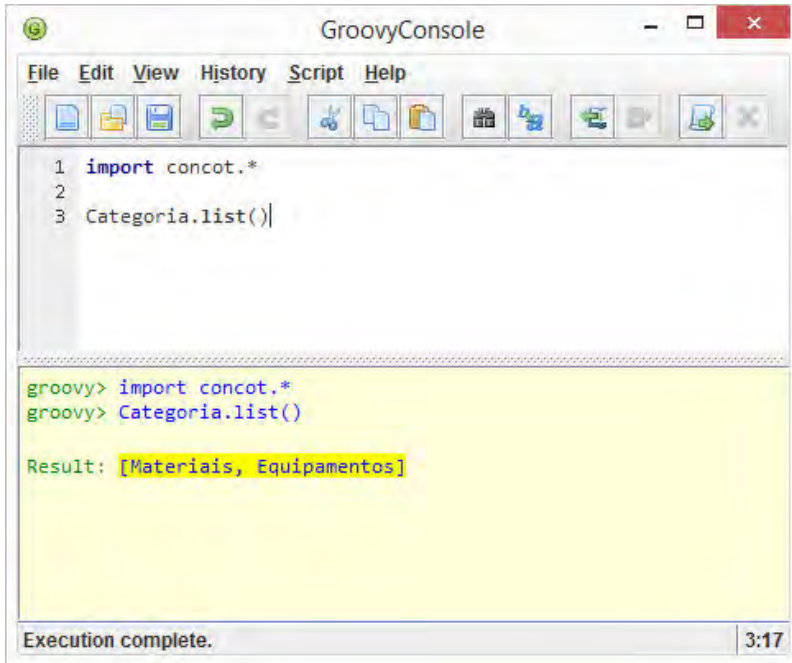



Fig. 6.1: 'Grails Console' em ação

Para facilitar sua vida neste capítulo, dado que todos os exemplos serão executados contra as classes do *ConCot*, sugerimos que seja incluída a instrução de importação de pacotes exatamente como no código a seguir:

```
import concot.*
```

6.2 Finders dinâmicos

O primeiro contato com os finders dinâmicos (*dynamic finders*) do GORM normalmente também é um daqueles momentos em que entendemos os ganhos que uma linguagem dinâmica como Groovy pode oferecer. De todas as opções de pesquisa oferecidas pelo Grails é a mais simples, mas não por isto a menos usada, pelo contrário: na geração de CRUD feita pelo recurso de *scaffolding* do Grails é inclusive o único modo de obtenção de dados adotado. Bom, mas com o que se parece esta criatura?

Assim como os métodos de persistência que vimos no capítulo 5 como `save()` e `delete()`, os finders dinâmicos parecem métodos estáticos incluídos em nossas classes de domínio. Da mesma forma, estes não existem diretamente no código-fonte, mas são gerados dinamicamente durante a execução da aplicação. Como veremos, o GORM irá interpretar o nome dos métodos que invocamos e, seguindo algumas convenções que veremos nesta seção, vai executar buscas no nosso banco de dados encontrando (ou não) aquilo que desejamos.

Obtendo uma única instância

Em diversos controladores gerados pelo scaffolding do Grails, iremos encontrar *actions* (ações) similares ao código a seguir:

```
def get(long id) {
    Categoria categoriaInstance = Categoria.get(id)
}
```

Apenas para lembrar, a definição da classe de domínio `Categoria` é bastante simples como podemos ver a seguir:

```
class Categoria {
    String nome
    // constraints omitidas para simplificar
}
```

Quando iniciado, o GORM irá automaticamente incluir alguns métodos em todas as nossas classes de domínio que servem para verificar a existência de uma entidade ou obter uma instância sua usando como parâmetro de busca sua chave primária/identificador.

Destes métodos, o mais usado é o `get`, que recebe como parâmetro um valor que deverá corresponder ao identificador da classe de domínio. No script a seguir podemos ver um exemplo de uso:

```
// Busca uma instância de Categoria com id 1
Categoria comId1 = Categoria.get(1)
```

Caso o registro não exista no banco de dados, `null` será o resultado da invocação deste método. Há ainda mais uma opção interessante para este

tipo de consulta. Estamos falando da função `read`, cujo uso é praticamente idêntico ao do `get` conforme vemos no exemplo a seguir:

```
// Busca uma instância de Categoria com id 2
Categoria comId2 = Categoria.read(2)
```

A diferença entre `get()` e `read()`? Basicamente `get` nos aponta uma armadilha que costuma assustar diversos programadores inexperientes com o funcionamento essencial do Hibernate.

Fig. 6.2: Atenção redobrada!

No capítulo 5.7 mencionamos rapidamente o conceito de sessão do Hibernate. Apenas para relembrar, trata-se de um cache local, definido em escopo de thread que armazena o estado de todas as nossas classes de domínio (ou entidades, se você preferir) para que sejam persistidos no banco de dados. Naquele momento falamos sobre o conceito de *flush*, no entanto o que não foi mencionado é que **o Hibernate pode persistir o estado dos nossos objetos mesmo que não invoquemos o método `save()`.**.

A chave para melhor entender como isto ocorre está no momento em que a instância de uma classe de domínio vai parar na sessão do Hibernate. Este "*ir parar na sessão do Hibernate*" tem um nome técnico: ser anexado (em inglês, *attached*) à sessão. Uma vez anexado, dizemos que o Hibernate passa a gerenciar o estado deste objeto, persistindo-o no banco de dados no momento em que os comandos SQL serão transmitidos ao SGBD. Quando um objeto é anexado a uma sessão do Hibernate?

- Ao obtermos o objeto através de uma consulta.
- Se persistimos o objeto usando o método `save()`.
- Quando manualmente o incluímos na sessão.

O que nos interessa agora é o primeiro caso. Por padrão ao alterarmos o valor de qualquer atributo mapeado o objeto será marcado para que seu estado seja persistido no banco de dados. E sabem o que é mais legal? Por padrão, isto ocorrerá no momento em que a sessão for fechada. Este comportamento é bastante sutil, sendo assim nada melhor que um script simples para ilustrá-lo:

```
import concot.*

Categoria comId1 = Categoria.get(1)
comId1.nome = "Nada pode dar errado"
```

Ao sair do escopo do script a sessão é fechada e o flush ocorre. Você deve ter notado que o atributo foi alterado mas que em momento algum chamamos o método `save()`. Intuitivamente, podemos pensar que o nome anterior do registro se manteve, porém executando a consulta a seguir no MySQL nota-se que este não foi o resultado esperado.

```
mysql> select * from categoria where id = 1;
+----+---------+----------------------+
| id | version | nome                 |
+----+---------+----------------------+
|  1 |       4 | Nada pode dar errado |
+----+---------+----------------------+
1 row in set (0.00 sec)
```

Este comportamento de detecção de alteração de estado e eventual persistência das mudanças no banco de dados também tem um nome técnico: *detecção de sujeira* (*dirt detection*). Dizemos que um objeto gerenciado pelo Hibernate está "sujo" quando seu estado foi alterado e ainda não foi persistido no banco de dados. Há inclusive uma função anexada a todas as nossas classes de domínio chamada `isDirty()` que retorna `true` caso o objeto se encontre neste estado. No script a seguir podemos ver um exemplo de sua aplicação.

```
import concot.*

Categoria comId = Categoria.get(1)
```

```
println comId.isDirty() // imprimirá 'false', nada foi alterado ainda
comId.nome = "Outro nome"
println comId.isDirty() // imprimirá 'true', o estado foi alterado e ainda
```

A função `read()` na realidade irá apenas desabilitar a detecção de sujeira na obtenção das entidades. Assim, o objeto só será persistido de fato no banco de dados quando explicitamente chamarmos o método `save()`. Podemos ver um exemplo de sua aplicação no script a seguir:

```
import concot.*

Categoria comId = Categoria.read(1)
println comId.isDirty() // imprimirá 'false', pois a detecção de sujeira
comId.nome = "Qualquer coisa"
println comId.isDirty() // imprimirá 'false', pois a detecção de sujeira
```

Lembre-se: `read()` retorna o objeto sem detecção de sujeira, no entanto você ainda poderá alterar o estado do seu objeto e persisti-lo executando o método `save()` sem problemas.

Finalmente, nem sempre queremos obter uma instância, mas sim apenas saber se ela existe. Nesses casos usamos a função `exists()`, que recebe como parâmetro o identificador da entidade e nos retornará um valor booleano indicando ou não sua existência. Um exemplo simples de sua aplicação pode ser visto no script a seguir:

```
/* Imagine que exista uma categoira com id = 1 */
println Categoria.exists(1) // imprimirá 'true'
/* Em nosso banco de dados imaginário não existem registros com identific
println Categoria.exists(-1) // imprimirá 'false'
```

Listando registros

Uma operação bastante comum na implementação de toda página de CRUD é a listagem de registros. Para facilitar nossa vida, o GORM injeta em todas as classes de domínio a função `list()` cujo uso mais simples, sem qualquer parâmetro, retorna todas as entidades armazenadas em uma tabela tal como no exemplo a seguir:

```
import concot.*

Categoria.list() // retorna todas as categorias cadastradas
```

Claro que, conforme o número de instâncias aumente, não é interessante sempre retornar todos os registros. Sendo assim há alguns parâmetros que podemos passar para esta função que a tornam perfeita para a implementação de telas de listagem nas quais desejemos paginar os dados. Segue a lista de parâmetros:

- `max` número máximo de instâncias a ser retornado.
- `offset` a posição a partir da qual os itens deverão ser listados.
- `order` se os registros deverão vir ordenados em ordem ascendente ou descendente. Aceita apenas dois tipos de valor: `asc` e `desc`. Caso nenhum seja fornecido, o padrão é `asc`.
- `sort` qual o atributo que será usado na ordenação das instâncias.
- `readOnly` desabilita a checagem de sujeira. Funciona exatamente como a função `read()`. O valor padrão deste método é `false`.
- `fetchSize` define o tamanho do *fetch*, ou seja, em um nível mais baixo, quantos registros por vez o driver JDBC deverá receber durante a consulta SQL. Muito útil no caso de otimizações.
- `timeout` quanto tempo em segundos deverá ser levado em consideração na política de timeout da consulta. Com isto, caso haja uma demora excessiva na obtenção das instâncias uma exceção será disparada.

A seguir podemos ver alguns exemplos de aplicação da função `list()`.

```
// obtendo todas as categorias
def todas = Categoria.list()
// obtendo apenas as dez primeiras instâncias
def primeiras10 = Categoria.list(max:10)
// obtendo apenas as dez primeiras instâncias
// ordenadas pelo nome em ordem descendente
```

```
def primeirasDezInvertidas = Categoria.list(max:10, sort:'nome', order:'d
// agora, aplicando o offset para obter a partir da posição 11
def primeirasDezInvertidas11 = Categoria.list(max:10, offset:11, sort:'no
```

Expressões de método

O aspecto mais interessante dos *finders* dinâmicos que são as expressões de método (*method expressions*), que nos permite escrever consultas de uma forma bastante simples tirando máximo proveito da natureza dinâmica do Groovy. Toda invocação de método em uma classe de domínio GORM é antes interceptada. Neste ato, o nome do método a ser executado é analisado e, se bater com algumas regras definidas pelo GORM, uma nova consulta será gerada e enviada para o SGBD. A seguir podemos ver um exemplo bem simples da aplicação deste princípio ao buscarmos por uma instância de `Categoria` cujo atributo `nome` seja `Materiais`.

```
/*
    Retornará, caso exista no banco de dados,
    uma instância de Categoria cujo atributo
    nome seja "Materiais"
*/
def categoriaMateriais = Categoria.findByNome("Materiais")
```

Toda *method expression* deve começar com o prefixo `findBy` ou `findAllBy`, usados respectivamente para obter como resultado apenas uma ou uma lista de instâncias. Seguido a este prefixo concatena-se o nome do atributo sobre o qual será feita a consulta. Também podemos aplicar operadores lógicos e comparadores com o objetivo de criar consultas mais complexas. Na documentação oficial do Grails [4] há um esquema da sintaxe a ser aplicada em uma *method expression* que iremos dissecar até o restante desta seção com uma série de exemplos que podemos ver a seguir:

```
ClasseDominio.[findBy|findAllBy]([Propriedade][Comparador]?[Operador bool
                                ([Propriedade][Comparador])
```

Que se inicie a dissecação. A propriedade diz respeito ao nome do atributo da classe de domínio que será usado na confecção da consulta. Não

há muito o que ser dito a seu respeito, apenas que deve ser digitado no formato *camel case*, sempre se lembrando de colocar a primeira letra do nome em maiúscula. O próximo elemento que nos interessa será o comparador. Na lista a seguir podemos ver a lista dos comparadores disponibilizados pelo GORM:

* `InList` encontra-se na lista passada como parâmetro. * `LessThan` menor que. * `LessThanEquals` menor ou igual a. * `GreaterThan` maior que. * `GreaterThanEquals` maior ou igual a. * `Like` equivale ao `like` do SQL. Usado para fazer busca por similaridade. * `ILike` similar a `ILike` só que não é `case sensitive`. * `NotEqual` diferente de. * `InRange` dentro dos limites superior e inferior fornecidos. * `RLike` usado para expressões regulares similar ao `Regexp LIKE` do MySQL. * `Between` entre dois valores. * `IsNotNull` valor não é nulo. * `IsNull` valor é nulo.

O leitor deve observar que na listagem de comparadores não há algum para igualdade. A razão para isto é simples: este é o comparador padrão caso nenhum outro seja definido. Vamos treinar um pouco? A seguir está a lista de algumas consultas que podemos fazer apenas com comparadores:

```
// A busca por igualdade de nome, que já conhecemos
Categoria.findByNome("Materiais")

// Busca todas as categorias com nome nulo
// apenas como exemplo (ignore as restrições que definimos para esta classe
Categoria.findAllByNomeIsNull()

// Agora, buscando todas as categorias que não
// possuam o nome nulo
Categoria.findAllByNomeIsNotNull()
```

Apenas para que possamos treinar com outros comparadores, vamos nos lembrar da definição da classe `Cotacao`, que pode ser vista na listagem a seguir:

```
class Cotacao {
    BigDecimal valor
    Date data
```

```
    static belongsTo = [item:Item,
                        moeda:Moeda,
                        fornecedor:Fornecedor]
}
```

Agora vamos fazer mais algumas consultas, apenas para treinar usando comparadores numéricos:

```
// Todas as cotações com valor maior que 1000
Cotacao.findAllByValorGreaterThan(1000)

// Usando between: cotações com valor entre 100 e 400
Cotacao.findAllByValorBetween(100,400)
```

Acredito que com estes exemplos o leitor já deve ter pegado parte do jeito da coisa. É chegado o momento de pensarmos em consultas um pouco mais complexas usando operadores lógicos. No caso dos finders dinâmicos há dois:

- *And* operador e
- *Or* operador ou

Imagine que desejemos buscar todas as cotações que tenham valor superior a **1000 reais**. Agora usaremos dois atributos e o operador lógico *And*. Como ficaria? O script a seguir nos dá um exemplo:

```
/* Primeiro buscamos a moeda "Real"
   Repare que usei 'findBy' pois queremos apenas um registro */
def real = Moeda.findByNome("Real")
// Agora, finalmente, a consulta completa
def cotacoes = Cotacao.findAllByMoedaAndValorGreaterThan(moeda, 1000)
```

Ou, se quisermos simplificar um pouco:

```
def cotacoes = Cotacao.findAllByMoedaAndValorGreaterThan(
        Moeda.findByNome('Real'), 1000)
```

Atenção especial deve ser dada ao comparador `like`. Imagine que tentemos buscar todas as categorias que tenham "ter" no nome e em nosso banco de dados há pelo menos um registro com o valor "Material" cadastrado. A consulta a seguir retornaria uma lista vazia:

```
Categoria.findAllByNomeLike("mat")
```

Isso porque precisamos usar o caractere especial % para denotar a expressão, tal como nos exemplos a seguir:

```
//Todas as categorias que comecem com "Mat"
Categoria.findAllByNomeLike("Mat%")

//Todas as categorias cujo nome termine em "ial"
Categoria.findAllByNomeLike("%ial")

//E finalmente, todas as categorias que tenham "ate" no nome
Categoria.findAllByNomeLike("%ate%")
```

Claro, também podemos paginar os resultados. Como fazer isso? Passando como último parâmetro para a consulta um objeto do tipo `Map` cujas chaves correspondam aos parâmetros que apresentamos na descrição da função `list()` 6.2. A seguir, novamente, alguns exemplos:

```
// Buscando apenas os 10 primeiros registros na busca por
// categorias que tenham "a" no nome
Categoria.findAllByNomeLike("%a%", [max:10])

// Agora, com offset
Categoria.findAllByNomeLike("%a%", [max:10, offset:11])
```

Persistindo com finders dinâmicos

Você também pode usar finders dinâmicos para gerar novos registros no seu banco de dados caso nenhuma instância seja encontrada. Como? Usando o prefixo `findOrSaveBy` e executando sua consulta. Observe o exemplo a seguir:

```
def equipamentos = Categoria.findOrSaveByNome("Equipamentos")
```

Caso não exista uma categoria com o nome "Equipamentos", em vez de retornar `null` o método `findOrSaveBy` irá criar um novo registro no banco de dados e nos retornar uma instância que represente aquele dado recém-criado.

Quando usar finders dinâmicos?

Finders dinâmicos são a melhor opção quando precisamos escrever consultas simples envolvendo um número pequeno de atributos (em minha experiência, três no máximo). O leitor deve levar em consideração também uma limitação importante deste recurso: todos os atributos envolvidos na consulta obrigatoriamente devem estar contidos na classe de domínio sobre a qual estamos executando a consulta.

É fácil mostrar como uma consulta com finders dinâmicos pode se mostrar problemática. Basta tentar interpretar a consulta a seguir com quatro atributos:

```
def real = Moeda.findByNome("Real")
def item = Item.get(4)
Cotacao.findAllByMoedaAndItemAndValorGreaterThanOrDataIsNull(moeda,item,1
```

Criteria

6.3 Criterias

Se finders dinâmicos são uma boa alternativa para consultas simples, as *criterias* nos oferecem uma poderosa ferramenta quando precisamos escrever consultas mais complexas em uma base de dados relacional e, ainda mais interessante, fornecendo um feedback visual ao programador que facilita bastante a leitura do código que escrevemos.

Caso o leitor já tenha alguma experiência com Hibernate o nome lhe soará familiar. Não é para menos: as criterias do GORM são na realidade uma DSL que por baixo dos panos efetua chamadas à API Criteria do Hibernate. Talvez você esteja curioso a respeito da aparência deste recurso, sendo assim, sem muita demora, eis um modo de usá-lo:

```
import concot.*

def criteria = Categoria.createCriteria()

criteria.get {
```

```
    eq 'nome', 'Materiais'
}
```

Estes comparadores podem também ser escritos usando parênteses, afinal de contas, são apenas funções Groovy normais:

```
import concot.*

def criteria = Categoria.createCriteria()

criteria.get {
    eq('nome', 'Materiais')
}
```

Qual forma usar? Trata-se de uma questão simplesmente estética.

Esta é a forma mais verbosa de se criar uma criteria. Ela foi escolhida apenas por ser mais didática neste momento e nos apresentar uma série de aspectos sobre o recurso. A função `createCriteria()` é injetada pelo GORM em todas as classes de domínio da sua aplicação e, quando chamada, nos retorna um objeto do tipo `grails.orm.HibernateCriteriaBuilder`. Repare no sufixo: sim, as criterias são uma aplicação do conceito de builder do Groovy que vimos neste livro quando tratamos da linguagem.

Por ser um builder, a construção de consultas se torna algo natural dado à natureza hierárquica que envolve toda expressão de busca. Ao executar a função `get` da nossa criteria, passamos como parâmetro um bloco de código que consistirá na DSL implementada pela equipe de desenvolvimento do GORM que executará os comandos contra a API Criteria do Hibernate. No caso, estamos usando o comparador de igualdade (`eq`), que sempre receberá dois parâmetros: o primeiro é o nome do atributo sobre o qual montaremos a consulta, e o segundo o valor usado na seleção dos dados.

Criteria, critério ou critérios?

Ainda não encontrei uma tradução satisfatória para este termo em português. A literal seria "Critérios", o problema é que na prática você irá usar o termo "criteria" o tempo inteiro. Sendo assim, optei por usar um neologismo fosse o mais próximo possível do seu dia a dia: "criteria" ou "criterias".

É claro que o único comparador não é o de igualdade. A seguir está uma lista com alguns dos principais comparadores providos pelo GORM e que usaremos em alguns exercícios práticos logo na sequência. Você pode obter uma lista completa na documentação oficial do Grails [3]. É importante ressaltar que quase todos apresentarão o mesmo funcionamento: o primeiro parâmetro corresponde ao nome do atributo sobre o qual é feito a busca e os restantes, quando há, referenciam os valores adotados em nossa consulta.

- `between` o valor da propriedade se encontrará entre dois valores. Exemplo: `between('valor', 10, 100)`
- `eq` igualdade. Exemplo: `eq('nome', 'Materiais')`
- `eq (case insensitive)` é possível desabilitar o case sensitive de equals. Para tal, basta incluir um parâmetro a mais no final da execução como no exemplo a seguir: `eq('nome', 'materiais', [ignoreCase:true])`
- `ne` diferente de. Exemplo: `ne('nome', 'Materiais')`
- `eqProperty` quando o valor de uma propriedade deve ser igual ao de outra. Exemplo: `eqProperty('sobrenome', 'nome')`
- `neProperty` uma propriedade possui valor diferente da outra. Exemplo: `neProperty('nome', 'sobrenome')`
- `gt` o valor deve ser maior que o parâmetro. Variações: `gt` (maior ou igual), `lt` (menor que), `lte` (menor ou igual) Exemplo: `gt('valor', 1000)`

- `gtProperty` o valor de uma propriedade deve ser maior que o de outra propriedade. Variações: `ltProperty` (menor que a propriedade), `gteProperty` (maior ou igual propriedade), `lte` (menor ou igual a propriedade). Exemplo: `gtProperty('valorVenda', 'valorCompra')`
- `idEq` usado quando se deseja fazer uma busca por identificador. Exemplo: `idEq(1)`
- `ilike` expressão do tipo `like` só que case insensitive. Exemplo: `ilike('nome', '%a%')`
- `in` o valor da propriedade deve estar entre aqueles presentes em uma lista. Atenção ao modo como o digitamos, pois *in* é uma palavra reservada do Groovy. Exemplo: `'in'('nome', ['Materiais', 'Equipamentos'])`
- `isEmpty` usado para buscar registros nos quais uma de suas propriedades, do tipo lista (um hasMany) está vazia. Exemplo: `isEmpty('itens')`
- `isNotEmpty` oposto de `isEmpty`. Exemplo: `isNotEmpty('itens')`
- `like` instrução do tipo `like`, exatamente como vimos no caso dos finders dinâmicos. Exemplo: `like('nome', '%a%')`
- `sizeEq` usada quando o número de itens em um relacionamento do tipo `hasMany` é igual ao valor passado como parâmetro. Há variações: `sizeGt` (tamanho maior que), `sizeGe` (tamanho maior ou igual), `sizeLt` (tamanho menor que), `sizeLe` (tamanho menor ou igual). Exemplo: `sizeEq('itens', 3)`
- `sqlRestriction` falaremos com mais detalhes sobre este item mais tarde. Basicamente ele nos possibilita enriquecer nossas consultas com instruções SQL nativas. Exemplo: `sqlRestriction("char_length(nome) < 30")`

Pela lista de comparadores fica nítido o poder das criterias e o quão mais avançado é este recurso em relação aos finders dinâmicos que vimos agora há pouco. Assim como fizemos naquele caso, vamos novamente nos exercitar um pouco com algumas consultas bem simples por enquanto, mas antes, é importante mostrar uma outra forma de se declarar criterias. Uma maneira bem mais simples, tal como podemos ver a seguir:

```
def categorias = Categoria.withCriteria {

}
```

A função `withCriteria` já cria uma criteria para nós e retorna a lista de resultados. Se não incluirmos nenhuma instrução na definição da consulta o resultado será todas as instâncias da classe de domínio. Há outras formas também:

```
def criteria = Categoria.createCriteria()

// obtendo uma lista com o resultado
criteria.list {

}

// obtendo uma lista com ainda menos código
criteria {

}

/*
    E se quisermos obter apenas um item na
    nossa consulta? Simples. :)
    Se a consulta retornar mais de um item como
    resultado, um erro será disparado, sendo assim
    cuidado na escrita do seu código.
*/

criteria.get {
    eq 'nome', 'Materiais'
}
```

Qual forma usar fica por conta da sua comodidade. Mais à frente quando formos falar de criterias desanexadas veremos como reaproveitar nossas consultas. Apenas para treinar um pouco, a seguir podemos ver mais um exemplo de consulta envolvendo três atributos na classe `Cotacao`.

```
import concot.*

/*
    com mais de um atributo
    Todas as cotações feitas nos últimos 10 dias
    e com valor maior ou igual a 1000, que não
    tenham o atributo fornecedor nulo mas que
    tenham a moeda nula
*/
def cotacoes = Cotacao.withCriteria {
    le 'data', new Date() - 10
    ge 'valor', 1000
    isNotNull 'fornecedor'
    isNull 'moeda'
}
```

Consultando associações

Os atributos de uma consulta Criteria não precisam ser todos da mesma classe de domínio. Imagine que desejemos buscar todas as cotações com valor superior a R$ 1000,00. Usando finders dinâmicos precisaríamos de duas consultas, com criterias, apenas uma como podemos ver adiante:

```
def cotacoes = Cotacao.withCriteria {
    gt 'valor', 1000
    moeda {
        eq 'nome', 'Real'
    }
}
```

Basta criarmos um novo bloco dentro da criteria que possua o mesmo nome que a associação presente em nossa classe de domínio. A estrutura hierárquica na qual escrevemos nosso código inclusive ajuda a tornar mais fácil de entender aquilo que desejamos. Apenas para treinar um pouco, que tal

buscar todas as cotações com valor superior a R$ 1000,00 e cujos itens cotados sejam pertencentes à categoria `Materiais`?

```
def cotacoes = Cotacao.withCriteria {
    gt 'valor', 1000
    moeda {
        eq 'nome', 'Real'
    }
    item {
        categoria {
            eq 'nome', 'Materiais'
        }
    }
}
```

Conjunções, disjunções e negações

Todas as consultas que vimos até este momento foram simples conjunções. Ao executarmos uma consulta como a seguir:

```
def cotacoes = Cotacao.withCriteria {
    gt 'valor', 1000
    moeda {
        eq 'nome', 'Real'
    }
}
```

Na realidade estamos executando algo como `todas as cotações com valor maior que 1000 **E** cuja moeda tenha como nome 'Real'`. A grande questão é: como usar uma disjunção (o famoso "ou")? Deste modo:

```
def cotacoes = Cotacao.withCriteria {
    or {
        gt 'valor', 1000
        lt 'valor', 100
    }
}
```

Assim obtivemos todas as cotações cujo valor seja superior a 1000 ou inferior a 100. Podemos usar a disjunção de forma explícita também:

```
def cotacoes = Cotacao.withCriteria {
    and {
        gt 'valor', 1000
        moeda {
            eq 'nome', 'Real'
        }
    }
}
```

Para finalizar, também podemos executar uma negação. Imagine que queremos negar um bloco inteiro dentro de uma criteria. Para tal basta usar a negação como no exemplo a seguir:

```
def cotacoes = Cotacao.withCriteria {
    not {
        gt 'valor', 1000
        moeda {
            eq 'nome', 'Real'
        }
    }
}
```

Nos últimos três últimos exemplos acredito que tenha fortalecido ainda mais a sua impressão de que com criterias temos, sem sombra de dúvidas, uma das formas mais poderosas e flexíveis para escrever nossas consultas. Muito melhor que a concatenação de strings que não raro aparecem em nossos sistemas, não é mesmo?

Usando restrições SQL

Sempre surgem situações nas quais poderíamos escrever nossas consultas de uma forma mais simples se pudéssemos acessar a camada inferior do ORM, ou seja, se conseguíssemos tirar proveito do SQL executado pelo SGBD. Criterias nos permitem isso graças ao comparador `sqlRestriction`.

É um comparador bastante simples pois só requer um parâmetro: a instrução SQL que usaremos para filtrar o resultado. Imagine que seja do nosso interesse buscar apenas as categorias cujo nome possua quatro caracteres. Como faríamos isto? Assim:

```
def categorias = Categoria.withCriteria {
    sqlRestriction 'char_length(nome) = 4'
}
```

O único ponto que o leitor deve levar em consideração é o fato de que muitas vezes irá usar instruções específicas de um SGBD (no exemplo, usei uma instrução do MySQL) que podem não estar presentes em outras soluções do mercado.

Projeções

Se formos buscar por uma definição de projeções no contexto do Hibernate encontraremos frases como *"são um recurso usado para customizar o resultado de uma consulta*

. A grande questão é: customizar como? Na verdade, aplicamos projeções quando desejamos que o resultado da nossa consulta não seja simplesmente uma lista de instâncias, mas sim uma lista de tuplas contendo atributos da nossa classe de domínio ou o resultado de expressões como soma, média e outras. Confuso? Vamos a alguns exemplos.

Imagine que desejamos saber qual a soma do valor de todas as cotações que valham mais que R$ 1000,00. Como você faria? Assim:

```
def criteria = Cotacao.createCriteria()

criteria.get {
    gt 'valor', 1000
    moeda {
        eq 'nome', 'Real'
    }
    projections {
        sum 'valor'
    }
}

// será retornado um valor numérico apenas.
```

Se o bloco `projections` estiver contido em uma criteria o valor retornado deixa de ser uma lista de instâncias da classe de domínio e passa a ser

uma lista de listas. Cada uma das listas retornadas conterá em seu interior as projeções que definimos no bloco `projections`. Caso use a função `get` e a lista contenha apenas uma lista com um único valor nesta, será retornado apenas aquele item.

Voltando, imagine que não precisemos da cotação inteira, mas sim apenas de seu identificador e valor. Como faríamos?

```
def criteria = Cotacao.createCriteria()

criteria {
    gt 'valor', 1000
    moeda {
        eq 'nome', 'Real'
    }
    projections {
        property 'id'
        property 'valor'
    }
}

// Exemplo de retorno:
// [[1, 1000], [2, 1010], [3, 4030.40]]
```

A instrução `property`, que recebe como parâmetro o nome do atributo presente em nossa classe de domínio, determina que iremos retornar apenas o conjunto de propriedades que definimos nas projeções. A seguir está uma lista de projeções para que o leitor possa ter uma ideia do poder deste recurso. Uma lista completa pode ser encontrada na documentação oficial do Grails [3].

- `property` Retorna a propriedade da classe em que foi executada a consulta. Exemplo: `property('nome')`
- `distinct` Retorna apenas valores distintos, tal como a instrução de mesmo nome que usamos no SQL. Exemplo: `distinct('valor')`
- `avg` A média dos valores de uma coluna. Exemplo: `avg('valor')`.

- `sum` A soma dos valores presentes em uma coluna. Exemplo: `sum('valor')`.

Como minimizar o consumo de memória quando uma critéria nos retorna um número significativo de instâncias

No futuro, o sistema *ConCot* viu sua base de dados crescer significativamente. Imagine que um dia alguém quisesse ver todas as cotações de itens cujo nome contenha a letra "a", mas a base de dados já conta com **bilhões** de registros. Como você faria? Você usaria *scrollable results*!

Scrollable Results são uma funcionalidade bastante interessante contida no Hibernate para lidar com este problema. Essencialmente, ao invés de uma criteria nos retornar uma lista com todos os itens, ela irá nos retornar um *iterador* (objeto do tipo *ScrollableResults*) que nos permite navegar entre os registros. Podemos ver como tirar proveito desta funcionalidade na listagem a seguir:

```
def criteria = Cotacao.createCriteria()
def scroll = criteria.scroll {
    item {
        like 'nome', '%a%'
    }
}

while (scroll.next()) {
    // vou lendo uma instância por vez e não todas!
    def cotacao = scroll.get()
}
```

Os seguintes métodos encontram-se disponíveis neste objeto:

- `next()` obtém o próximo item do iterador e move para o próximo. Retorna `true` caso seja possível fazer esta movimentação.
- `first()` move para o primeiro registro.
- `last()` move para o último registro.
- `get()` retorna a instância corrente.

- `isLast()` retorna `true` caso estejamos na última instância do iterador.

Uma lista completa pode dos métodos presentes na classe `ScrollableResults` pode ser obtida na documentação oficial do Hibernate [9].

Fig. 6.3: Atenção redobrada!

Muita atenção com o driver Connector/J do MySQL: nem sempre ele funciona como gostaríamos. Segundo a API JDBC, os dados deveriam ser enviados para o cliente somente quando ao movimentarmos o cursor encontrássemos um momento no qual tivéssemos chegado ao fim da lista, certo? Infelizmente, o Connector/J apenas emula este comportamento, trazendo para o cliente todos os dados para a memória.

Caso esteja usando o MySQL, verifique a documentação atual do driver JDBC e consulte este post [29] no StackOverflow.

Criando criterias em tempo de execução

Uma vantagem do uso de criterias é o fato de podemos criá-las dinamicamente. Lembre-se que o que passamos na construção de uma consulta é na realidade código Groovy. Sendo assim, nada impede que possamos montar nossas consultas em tempo de execução. Um exemplo simples: imagine que no *ConCot* implementamos uma função que define se devemos buscar cotações por uma moeda específica ou não:

```
def buscarCotacoes(String nomeMoeda, BigDecimal valor) {
    Cotacao.withCriteria {
        eq 'valor', valor

        //Você pode incluir um if na construção de uma criteria
        if (nomeMoeda) {
```

```
            moeda {
                eq 'nome', nomeMoeda
            }
        }
    }
}
```

Na realidade, é possível usar qualquer estrutura de controle de fluxo Groovy na definição de uma criteria, o que torna as possibilidades praticamente infinitas. A seguir podemos ver um outro exemplo usando o `switch`:

```
def buscarCotacoes(int tipoItem, BigDecimal valor) {
    Cotacao.withCriteria {
        eq 'valor', valor
        // Usando switch para o mesmo fim
        switch (tipoItem) {
            case 1:
                item {
                    categoria {
                        eq 'nome', 'Materiais'
                    }
                }
            break
            case 2:
                item {
                    categoria {
                        eq 'nome', 'Equipamentos'
                    }
                }
                break;
            default:
                isNull 'item'
        }
    }
}
```

Para entender a vantagem deste recurso, o leitor deve se lembrar como seria o procedimento para se obter o mesmo resultado usando consultas SQL nativas: nesses casos é necessária a concatenação de strings, o que resulta em

código muito mais complexo e propenso a erros.

Criterias desconectadas

Criterias desconectadas (*detached criteria*) nos possibilitam reaproveitar consultas de uma forma bastante interessante. A principal diferença entre estas e a criteria convencional é o fato de não estar associada a nenhuma sessão do Hibernate.

Criar uma *Detached Criteria* é simples, basta executar instruções similares à exposta a seguir:

```
import concot.*
import grails.gorm.DetachedCriteria // obrigatório

def criteria = new DetachedCriteria(Cotacao).build {
    moeda {
        eq 'nome', 'Real'
    }
    gt 'valor', 1000
}
```

O construtor da listagem recebe como parâmetro uma classe de domínio gerenciada pelo GORM, enquanto o método `build` criará a criteria para nós. Ao ser criada nenhuma, consulta será executada, apenas teremos o objeto pronto para que possa ser reaproveitado em outros pontos do sistema.

E como executamos uma criteria desanexada? Executando os mesmos métodos `list` e `get` que vimos anteriormente:

```
import concot.*
import grails.gorm.DetachedCriteria // obrigatório

def criteria = new DetachedCriteria(Cotacao).build {
    moeda {
        eq 'nome', 'Real'
    }
    gt 'valor', 1000
}
```

```
//obtem a lista de cotações
def resultado = criteria.list()
```

Composição de criterias com Criterias Desconectadas

Muitas vezes você se verá escrevendo repetidas vezes o mesmo trecho em diferentes consultas em seu sistema. Nesses casos, criterias desconectadas podem lhe ajudar a obter uma melhor componentização. Para tal, segue um exemplo prático no *ConCot*. Sempre precisamos buscar cotações que estejam em Reais. Sendo assim, nada mais natural que criemos uma criteria desconectada para isto.

```
def buscaCotacaoEmReais = new DetachedCriteria(Cotacao).build {
    moeda {
        eq 'nome', 'Real'
    }
}
```

A consulta foi criada e está pronta agora para ser reaproveitada, tal como no exemplo a seguir:

```
def cotacoesCaras = Cotacao.withCriteria {
    buscaCotacaoEmReais //olha aqui a criteria desconectada :)
    gt 'valor', 1000000
}
```

Ao ser executado, o código na realidade vai corresponder ao seguinte:

```
def cotacoesCaras = Cotacao.withCriteria {
    moeda {
        eq 'nome', 'Real'
    }
    gt 'valor', 1000000
}
```

Processamento múltiplos registros com criterias desconectadas

É possível alterar múltiplos registros de uma única vez usando criterias desconectadas. Para tal, basta usarmos as instruções `updateAll` ou

deleteAll destes objetos para, respectivamente, editar ou excluir em massa uma gama de registros.

Dois exemplos ilustram bem essa funcionalidade: primeiro, vamos editar o valor de todas as cotações para zero caso sua categoria seja "Gratuito". A seguir podemos ver como isto é feito:

```
def criteriaCotacoes = new DetachedCriteria(Cotacao).build {
    item {
        categoria {
            eq 'nome', 'Gratuito'
        }
    }
}

criteriaCotacoes.updateAll([valor:0])
```

Ao método updateAll fornecemos um objeto do tipo Map no qual as chaves correspondam ao nome do atributo da classe de domínio a ser modificado e o valor, aquele que desejamos persistir no banco de dados.

Para excluir todos os itens gratuitos é igualmente simples como vemos a seguir:

```
def criteriaCotacoes = new DetachedCriteria(Cotacao).build {
    item {
        categoria {
            eq 'nome', 'Gratuito'
        }
    }
}
// Sem itens gratuitos daqui pra frente. :)
criteriaCotacoes.deleteAll()
```

Paginação e ordenação com criterias

Paginação de resultados com criterias é bastante simples. Basta usar as expressões a seguir no bloco da sua consulta:

- maxResults(int) define quantas instâncias serão retornadas pela criteria.

- `firstResult(int)` a partir de qual posição os dados serão expostos.

A listagem a seguir expõe um bom exemplo de como usar essas instruções:

```
/* Buscará 10 instâncias a partir da posição 11 */
def moedas = Moeda.withCriteria {
    maxResults(10)
    firstResult(11)
}
```

Ordenação é muito parecida com o que vimos nos finders dinâmicos. Basta usar a instrução `order`, que recebe dois parâmetros. O primeiro corresponde ao nome do atributo sobre o qual será feita a ordenação e o segundo se será em ordem crescente (`asc`) ou descendente (`desc`). O exemplo a seguir expõe o uso da instrução:

```
/* Buscará 10 instâncias a partir da posição 11
   ordenadas pelo nome em ordem decrescente */
def moedas = Moeda.withCriteria {
    maxResults(10)
    firstResult(11)
    order('nome', 'desc')
}
```

Quando uso criterias?

Como dito no início desta seção, opte pelo uso de criterias sempre que precisar escrever consultas mais complexas e cuja legibilidade é comprometida quando escrita em instruções SQL, HQL.

Se for observado que sempre é repetido o mesmo trecho em diferentes consultas, criterias também caem como uma luva, pois evitam a necessidade de concatenação de strings, que normalmente dão origem a código de altíssima complexidade e muito propensos a erros.

Além disso, como observado, são também uma excelente ferramenta quando precisamos fazer a edição ou exclusão de múltiplos registros em nossa base de dados de uma maneira simples.

6.4 Buscas por where (where queries)

Ao inserirem na versão 2.0 do Grails as criterias desconectadas a equipe de desenvolvimento não estava apenas buscando uma maneira mais interessante de componentização de consultas: queriam na realidade uma nova ferramenta de pesquisa que tivesse o mesmo poder que as criterias mas com uma sintaxe mais próxima do Groovy. Sim, estou falando das buscas por `where`.

A melhor forma de apresentar as buscas por `where` é comparando-as com a criteria. Na listagem a seguir podemos ver a diferença de forma nítida:

```
// A versão criteria de uma consulta que já usamos bastante
// neste capítulo
Cotacao.withCriteria {
    gt 'valor', 1000
    moeda {
        eq 'nome', 'Real'
    }
}

// E a versão por where
Cotacao.findAll {
    moeda.nome == 'Real' && valor > 1000
}
```

Buscas por `where` nos oferecem uma alternativa mais flexível que os finders dinâmicos e bem menos verbosa que a criteria **sem que tenhamos qualquer tipo de comprometimento**. Antes de nos aprofundarmos no assunto é importante primeiro entender **por que o nome where queries**.

```
Cotacao.where {
    moeda.nome == 'Real' && valor > 1000
}
```

A função `where` anexada a todas as classes de domínio pelo GORM recebe como parâmetro um bloco de código no qual digitamos nossa consulta com código Groovy e nos retorna uma criteria desconectada. Sendo assim, todas as funções que vimos serem usadas para atualização em massa de dados (`updateAll` e `deleteAll`) e scrollable results (`scroll`) também podem ser executadas contra este objeto.

Escrevendo consultas

Se você consegue escrever uma expressão booleana em Groovy baseada nos atributos das nossas classes de domínio já sabe como usar as buscas por `where`. Vamos começar por algumas consultas simples, que já vimos aqui escritas com criterias, agora usando `where`.

```
// Buscando a moeda Real
Moeda.find {
    nome == 'Real'
}
// Buscando agora as cotações com valor maior
// que 1000 reais
Cotacao.findAll {
    moeda.nome == 'Real' && valor > 1000
}
```

Enquanto a função `where` nos retorna uma criteria desconectada que não executa consulta alguma naquele momento (este comportamento é chamado *lazy*), as funções `find` e `findAll` retornam, respectivamente, um único resultado ou uma lista. Claro que podemos usar conjunções (*and*) e disjunções (*or*) em nossas consultas. Para tal, basta agruparmos os componentes da consulta usando algo familiar a nós: parênteses.

```
// Usando conjunção
// Moeda = Real E valor > 1000
Cotacao.findAll {
    (moeda.nome == 'Real') && (valor > 1000)
}

// Usando uma disjunção
// Com valores maior que 1000 ou menores que 100
Cotacao.findAll {
    (valor < 100) || (valor > 1000)
}

// Misturando disjunções e conjunções
Cotacao.findAll {
    (moeda.nome == 'Real') && ( (valor < 100) || (valor > 1000) )
```

```
}

// Se quiser, também pode usar os métodos da
// criteria desconectada. Lembra deles?
def consulta = Cotacao.where {
    (moeda.nome == 'Real') && (valor > 1000)
}
def resultado = consulta.list(sort:'valor')
```

Não se esqueça também da negação. Se quisermos todas as cotações que não foram feitas em Real escreveríamos algo como:

```
def tudoMenosReal = Cotacao.findAll {
    ! (moeda.nome == 'Real')
}
```

Como pode ser visto, temos uma DSL extremamente natural ao programador: uma que você já conhece. Pense em uma expressão booleana, escreva-a e pronto: sua consulta está pronta. E como você lida com relacionamentos? A primeira parte você já viu, lembra?

```
Cotacao.findAll {
    moeda.nome == 'Real'
}
```

Basta que você referencie o nome da associação na sua consulta e pronto! Os operadores de comparação também funcionam exatamente como você esperaria que se comportassem em Groovy: `==` (igual), `!=` (diferente), `>` (maior), `>=` (maior ou igual), `<` (menor), `<=` (menor ou igual). Mas há alguns que não são tão intuitivos: três apenas.

- `in` o valor se encontra dentro de uma lista
- `==~` like case sensitive
- `=~` like case insensitive

```
// Todas as moedas que contenham e no nome.
// Retornará o 'Real'
```

```
Moeda.findAll {
    nome ==~ "%e%"
}

//Todas as moedas cujo nome seja 'Real' ou 'Dolar'
Moeda.findAll {
    nome in ['Real', 'Dolar']
}
```

É também possível escrever consultas com o operador `between`. Como? Usando *ranges* como no exemplo a seguir:

```
// Todas as cotações com valor entre 100 e 1000
Moeda.findAll {
    valor in 100..1000
}
```

Nulidade? Fácil também. Basta comparar o atributo com `null`.

```
// Todas as cotações que não tenham fornecedor
Cotacao.findAll {
    fornecedor == null
}
```

E comparação entre propriedades?

```
// Todos os itens com valor de compra igual ao de venda
Cotacao.findAll {
    valorCompra == valorVenda
}
```

Também podemos executar comparações relacionadas ao tamanho de uma associação do tipo `hasMany` tal como a seguir:

```
// todas as categorias que tenham mais de 0 itens
// associados
def itens = Categoria.findAll {
    itens.size() > 0
}
```

Consultas compostas

Se no fundo uma busca `where` é uma criteria desconectada, nada impede que também possamos tirar proveito daquela componentização que vimos antes. Como?

```
// A consulta por moeda 'Real'
def buscaReal = Cotacao.where {
    moeda.nome = 'Real'
}

// Compondo a consulta, agora por valor
def buscaValor = consultaReal.where {
    valor > 1000
}

def resultado = buscaValor.list()
[code]

A segunda consulta é como se tivéssemos escrito

[code groovy]
def buscaValor = Cotacao.where {
    moeda.nome == 'Real' && valor > 1000
}
```

Subconsultas

Não há projeções em consultas `where`, mas há algo interessante que entra em seu lugar: são as subconsultas, que nos permitem escrever consultas com um grau maior de inteligência através da inclusão de funções especiais dentro de uma consulta `where`. Para melhor entender este recurso, vamos listar estas funções:

- `avg` retorna o valor médio de todos os itens em uma consulta.
- `sum` retorna a soma do valor de todos os itens na consulta.
- `max` retorna o valor máximo do atributo na tabela.

- `min` retorna o valor mínimo na tabela.
- `count` quantos itens estão na tabela ou consulta.
- `property` retorna uma propriedade a partir da lista de propriedades.

Se eu quiser saber quais as cotações cujo valor seja maior que a média, como faço? Assim:

```
def cotacoes = Cotacao.findAll {
    valor > avg(valor)
}
```

E também podemos aplicar essas funções a subconsultas. Por exemplo: se quiséssemos que a consulta anterior fosse aplicada apenas aos itens cuja moeda é o Real. Para isso, usaríamos o método `of`, que recebe como valor uma outra consulta.

```
def cotacoes = Cotacao.findAll {
    valor > avg(valor).of {moeda.nome == 'Real'}
}
```

Há também algumas funções que você pode usar em suas consultas que são muito úteis:

- `second` Retorna apenas o segundo da propriedade caso ela seja do tipo `Date`.
- `minute` Retorna apenas o minuto de uma propriedade do tipo `Date`.
- `hour` A hora de uma propriedade do tipo `Date`.
- `day` O dia de uma propriedade do tipo `Date`.
- `month` O mês da propriedade do tipo `Date`.
- `year` O ano de uma propriedade do tipo `Date`.
- `upper` Converte o valor da propriedade para letras maiúsculas.
- `lower` Converte o valor da propriedade para letras minúsculas.

- `trim` Remove o caractere de espaço do início e do fim de uma string.
- `length` Retorna o tamanho de uma propriedade do tipo string.

Com estas funções podemos escrever consultas interessantes. Talvez como na função a seguir, que me retornará todas as minhas cotações do ano passado como parâmetro dentro de um dado intervalo de valores para uma moeda cujo nome também será passado como parâmetro:

```
def buscarCotacoes(String nMoeda, int ano, BigDecimal valorMinimo, BigDeci
    Cotacao.findAll {
        (year(data) == ano) &&
        (valor in (valorMinimo)..(valorMaximo)) &&
        (lower(moeda.nome) == nMoeda.toLowerCase())
    }
}
```

Quando uso buscas por where?

Como dito no início desta seção, buscas por `where` ficam entre criteria e finders dinâmicos. São uma excelente opção devido à sua legibilidade, porém como uma pequena limitação: não possuem projeções. Sendo assim, caso projeções sejam um requisito para seu código, criterias soam como uma melhor opção.

6.5 HQL

Talvez a opção mais flexível orientada a objetos que o GORM nos oferece seja o suporte a HQL (*Hibernate Query Language*), uma linguagem poderosíssima adotada pelo Hibernate que possui uma vantagem bastante interessante: é bastante parecida com SQL. Se você ainda não conhece HQL, mas já possui alguma experiência com SQL, pode-se dizer que, a grosso modo, em vez de lidarmos com tabelas e colunas manipularemos respectivamente objetos e suas propriedades.

Isso não quer dizer que seja uma linguagem de fácil aprendizado: neste livro iremos apenas expor o fundamental para que você possa começar a tirar proveito desta tecnologia. Para aprendê-la em profundidade (o que re-

comendo), sugiro a leitura da documentação oficial do Hibernate [8] que é bastante interessante.

Executando HQL

Em cada classe de domínio do seu projeto o GORM irá inserir dois métodos para que possamos executar consultas usando HQL: `findAll` e `find` que, respectivamente, nos retornam uma lista de objetos ou apenas um (quando sua consulta for escrita de tal forma que apenas um item seja retornado). Há diversas assinaturas para este método, mas a principal recebe apenas um parâmetro, nossa consulta, tal como exposto a seguir:

```
// Todas as categorias que tenham a letra A no nome
Categoria.findAll("from Categoria c where c.nome like '%a%'")

// A categoria cujo id é 1
Categoria.find("from Categoria c where c.id = 1")
```

Se você nunca trabalhou com HQL mas já viu SQL, não se assuste com a ausência da palavra-chave `select`. Falaremos mais sobre ela mais tarde. Repare que estamos dando um *alias* (*apelido*) à entidade `Categoria` apenas para facilitar a digitação das instruções. Ao invés de repetir a palavra `Categoria`, por que não simplificar digitando apenas *c*? Claro, o alias é opcional quando em nossa consulta só referenciamos uma entidade. Sendo assim, as consultas a seguir são igualmente válidas:

```
// Todas as categorias que tenham a letra A no nome
Categoria.findAll("from Categoria where nome like '%a%'")

// A categoria cujo id é 1
Categoria.find("from Categoria where id = 1")
```

Também podemos parametrizar nossas consultas. Afinal de contas não é interessante ficar concatenando strings quando queremos variar os valores usados em nossas pesquisas, não é mesmo? Vejamos alguns exemplos. Primeiro, com parâmetros posicionais, como no exemplo a seguir:

```
import concot.*
```

```
Categoria.findAll("from Categoria where id = ?", [11])
```

O caractere `?` representa um parâmetro. A forma mais primitiva de passarmos parâmetros para nossas consultas se dá como no exemplo anterior. Nós os fornecemos no interior de uma lista na mesma ordem em que aparecem no texto de nossa consulta. É o que chamamos de *parâmetros posicionais*. A segunda forma, e mais interessante são os *parâmetros nomeados*:

```
import concot.*

Categoria.findAll("from Categoria where id = :id", [id:11])
```

A diferença é que agora, caso o parâmetro se repita em nossa consulta, como no código adiante, não precisamos fornecer o valor duas ou mais vezes, mas sim apenas uma.

```
import concot.*
/*
    Todos os itens cujo nome contenha em si
    o mesmo nome que sua categoria.

    (uma busca tola, apenas para fins didáticos)
*/
Item.findAll("from Item i where i.nome like '%:nome%' and \
    i.categoria.nome = :nome", [nome:'Material'])
```

A propósito, strings de múltiplas linhas do Groovy não funcionam com HQL, sendo assim, caso queiramos tirar proveito delas para facilitar a leitura do nosso código, precisamos digitá-las tal como no exemplo anterior. Aliás, neste mesmo exemplo aprendemos também a escrever consultas que lidem com associações. Reparou como referenciamos o relacionamento *categoria* (do tipo `belongsTo`) da classe `Item`?

Paginação e ordenação

Paginar consultas HQL é exatamente como fizemos no caso dos finders dinâmicos. São inclusive os mesmos parâmetros tal como podemos ver a seguir quando buscamos dez instâncias contando a partir da posição 11.

```
/*
 Simplesmente 'from Item' indica que iremos
 buscar todos os registros, pois não há filtragem
 de dados.
*/
Item.findAll("from Item", [max:10, offset:11])
```

Já a ordenação é feita como faríamos em SQL. A diferença está no fato de que iremos referenciar na ordenação as propriedades das classes referenciadas na consulta e não colunas:

```
Item.findAll(from Item where nome like '%a%' sort by nome",
             [max:10, offset:11])
```

Consultas para relatórios

Até agora só fizemos consultas que nos retornam instâncias da classe na qual invocamos o método `findAll` (ou `find`), mas há situações nas quais queremos obter apenas *parte* da classe de domínio (ou suas referências), tal como fizemos com as projeções nas criterias. Sim: a palavra-chave `select` volta a ter uso aqui.

Imagine que desejemos implementar um relatório de cotações que contenha os seguintes campos:

- Identificador da cotação
- Categoria da cotação (apenas seu nome)
- Nome do item cotado
- Moeda (apenas o nome)
- Valor da cotação

Este relatório receberá como parâmetro apenas a data de início e fim, que usaremos na composição da nossa função cuja implementação podemos ver a seguir:

```
def cotacoesPorData(Date dataInicio, Date dataFim) {
  Cotacao.findAll("select c.id, c.item.categoria.nome, \
      c.item.nome, c.moeda.nome, c.valor \
  from Cotacao c \
    where c.data between :inicio and :fim",
    [inicio:dataInicio, fim:dataFim])

}
```

Usamos a palavra-chave `select` na prática quando queremos customizar o que será retornado. Neste caso, não virá uma lista de instâncias da nossa classe de domínio, mas sim uma lista de matrizes de `Object` (`Object[]`), na qual a ordem no interior de cada matriz corresponderá à que declaramos em nossa consulta. As famosas funções agregadoras.

A vantagem de uma consulta deste tipo é a sua leveza. Em vez de buscarmos o grafo completo, obtemos apenas aquilo que interessa ao nosso relatório. Claro, as regras de paginação e ordenação também se aplicam a esse tipo de consulta pois queremos evitar estouros de memória conforme nosso banco de dados aumenta de tamanho.

Assim como nas criterias, também temos funções especiais que nos ajudam a implementar relatórios mais complexos. Na lista a seguir podemos ver algumas destas funções. Para uma lista completa, recomendo a leitura da documentação oficial do Hibernate sobre HQL [**?**].

- `avg` A média dos valores de um dado atributo da nossa classe de domínio. Exemplo: `avg(cotacao.valor)`.
- `sum` A soma dos valores de um atributo do domínio. Exemplo: `sum(cotacao.valor)`.
- `count` Conta as ocorrências de um atributo do domínio. Exemplo: `count(cotacao.moeda)`.

Um exemplo rápido de consulta usando estes operadores. Talvez queiramos obter a soma de todas as cotações feitas em Real.

```
Cotacao.find("select sum(c.valor) from Cotacao c\
 where c.moeda.nome = 'Real'")
```

Ou então, ainda mais interessante: a média de preços para determinado item.

```
COtacao.find("select avg(c.valor) from Cotacao c\
where c.item.nome = ?", ['Britador'])
```

Quando usar HQL?

HQL é uma alternativa interessantíssima quando precisamos escrever consultas um pouco mais complexas ou focadas na implementação de relatórios. Trata-se de uma linguagem de consulta bastante poderosa e que neste capítulo, devido à limitação de espaço, acabamos por tratar apenas do funcionamento básico. Sua única limitação aparece quando precisamos escrever consultas dinâmicas, situações nas quais o programador acaba precisando concatenar strings e, com isto, termina por produzir código difícil de manter, de alta complexidade e muito propenso a erros.

Capítulo 7

A camada web: controladores

Neste capítulo vamos entender o processo por trás de uma ação simples executada por qualquer usuário. Veremos como o ato de acessar uma URL do nosso sistema, por exemplo http://localhost:8080/concot/categoria, resulta na renderização de páginas como a exposta a seguir.

7.1 Mas antes vamos falar um pouco sobre scaffolding?

No capítulo 4 criamos alguns controladores mas não chegamos a ver como estes funcionam pois os que vimos foram gerados com o auxílio do *scaffolding*. Apenas para lembrar, veja como ficou a nossa classe `CategoriaController` naquele momento:

```
package concot

class CategoriaController {

    static scaffold = Categoria
}
```

Este atributo estático chamado `scaffold` recebe como valor a classe de domínio `Categoria` e será usado pelo Grails para, em tempo de execução,

gerar todo o código responsável por nos fornecer um CRUD essencial para o nosso sistema que nos permitirá incluir, alterar, editar e listar os nossos registros de categorias no banco de dados. Isso inclui tanto o controlador quanto as páginas GSP usadas na renderização.

Atributo scaffold

O atributo scaffold na realidade pode receber dois valores. O primeiro é a classe que desejamos usar para gerar o CRUD. A segunda é simplesmente o valor booleano `true`.

Quando passamos `true` como valor, o Grails irá usar como base para descobrir qual a classe de domínio o nome do controlador, tal como no exemplo a seguir, que gerará o scaffolding para a classe de domínio `Item`.

```
class ItemController {
    static scaffold = true
}
```

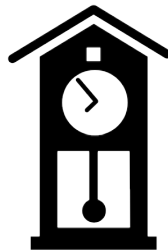



Fig. 7.2: Compatibilidade

Na versão 3.0 do Grails o scaffold dinâmico não foi incluído entre os recursos do framework, mas não se assuste: ele provávelmente estará de volta na versão 3.1 ou mesmo antes. :)

É importante que nos lembremos o que é o “scaffolding”. Traduzindo a palavra para o português temos o termo “andaime”. O que é um andaime? É

uma estrutura que nos possibilita obter acesso a alguma coisa ou lugar onde possamos nos escorar e, a partir dali, atingir algum objetivo. O termo tem sua origem na construção civil e, caso o nome lhe pareça familiar mas ainda não consiga se lembrar de ter visto um, a foto a seguir refrescará sua memória. :)

Fig. 7.3: Scaffolding

O que é este andaime que Grails nos oferece? A que desejamos ter acesso com ele? Como assim me escorar? Me escorar onde? O que desejamos construir com ele?

Não sei se você já reparou, mas algumas das tarefas que executamos durante o desenvolvimento de uma aplicação são bastante repetitivas. Pegue como exemplo o código que normalmente escrevemos para construir um simples CRUD. A impressão que tenho é a de que sempre executamos exatamente as mesmas tarefas:

1) Usuário preenche um formulário e o submete.

2) Obtemos os valores preenchidos pelo usuário após a submissão do formulário e os usamos para preencher os atributos de um ou mais objetos.

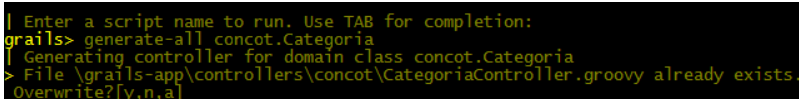
3) Validamos os valores para nos certificar de que foram corretamente preenchidos.

4) Tudo estando ok, persistimos nosso objeto de domínio e em seguida redirecionamos nosso usuário para uma página qualquer.

E você escreverá este código inúmeras vezes. É até possível tirar proveito da orientação a objetos e modularizar boa parte deste trabalho, evitando toda essa repetição, mas em sua essência o trabalho sempre será bastante próximo disto. Mais do que isso, não sei se é o seu caso, mas a esmagadora maioria dos programadores acha esse tipo de trabalho muito tedioso. Será que investimos tanto assim na nossa formação para, no final do dia, terminar fazendo essencialmente... isto???

Dado que é um trabalho repetitivo e que ocorre em praticamente todo projeto, não seria legal se houvesse algo como um "assistente" que o gerasse automaticamente para nós? Melhor ainda: e se este assistente gerasse para nós código fácil de ser customizado, de tal modo que nós só precisássemos alterar aquilo que realmente importa?

O que é este andaime? É a geração automática de todo código-fonte que execute tarefas repetitivas durante o desenvolvimento mas que nos permita customizá-lo conforme se mostre necessário. *A que desejamos ter acesso com este andaime?* Uma melhor produtividade a partir do momento em que podemos investir nosso tempo na implementação de partes mais complexas dos nossos sistemas e que **realmente importem e agreguem valor**. *Como assim me escorar?* Ele nos fornece todo o código-fonte inicial, é nele que me escoro, pois apenas preciso customizá-lo de acordo com as minhas necessidades. *O que desejamos construir com este scaffolding?* Todas (ou quase) as páginas de cadastro do nosso sistema!

E sabem o que é mais importante em andaimes? Depois que atingimos nosso objetivo podemos simplesmente nos esquecer deles ou mesmo jogá-los fora já que o trabalho está pronto e não precisamos mais deles. O scaffolding é isto: nós iremos usá-lo repetidas vezes até chegarmos ao ponto que desejamos. Depois disso você simplesmente esquece-se dele.

Até este momento nosso andaime é invisível: apenas declaramos um atributo estático e associamos uma classe a ele e o Grails gentilmente gera todo o código-fonte em tempo de execução (por isto o chamo de dinâmico). É uma opção muito interessante quando ainda estamos modelando nossas classes de domínio e queremos experimentar como se comportarão as páginas de CRUD da nossa aplicação, mas infelizmente não são uma boa alternativa quando precisamos customizar seu funcionamento.

Sendo assim apresento um novo comando ao leitor: `generate-all`, que recebe como parâmetro o nome completo da classe de domínio (o que inclui seu pacote). Este irá gerar um novo controlador para nós além de todas as páginas GSP que seriam geradas em tempo de execução pelo framework, o que nos permitirá agora customizá-las de acordo com nossas necessidades. Para começar nosso alvo será a classe mais simples do nosso sistema: `Categoria`. É o que chamo de *"scaffolding estático*

. Basta executar o comando a seguir:

```
grails generate-all concot.Categoria
```

O script irá nos perguntar se desejamos substituir o arquivo `CategoriaController` que já havíamos criado tal como exposto na imagem a seguir. Basta digitar "y" (*yes*) ou "a" (*all*) e em seguida pressionar `ENTER`.

```
| Enter a script name to run. Use TAB for completion:
grails> generate-all concot.Categoria
| Generating controller for domain class concot.Categoria
> File \grails-app\controllers\concot\CategoriaController.groovy already exists.
 Overwrite?[y,n,a]
```

Fig. 7.4: grails generate-all concot.Categoria

No diretório `grails-app/views/categoria` serão gerados cinco arquivos GSP: `_form.gsp`, `create.gsp`, `edit.gsp`, `index.gsp` e `show.gsp`. Correspondem às páginas que compõem o CRUD desta classe de domínio. Também será criado um novo `CategoriaController.groovy` no diretório `grails-app/controllers/concot`, desta vez bem maior que nossa primeira versão que continha apenas uma linha. Agora você pode customizar 100% do seu scaffolding.

Fiz este pequeno parêntese sobre scaffolding apenas para criar o material que usaremos no restante deste capítulo. Mais à frente falaremos mais sobre como você pode customizar a geração de código com Grails, mas por enquanto só nos interessa a classe `CategoriaController`.

7.2 Entendendo os controladores

O que é um controlador? Nós já falamos sobre o que é o padrão de projeto MVC 4.4, mas como a equipe responsável pelo desenvolvimento do Grails define um controlador? Definem como o responsável por lidar com as requisições recebidas pela aplicação e preparar a resposta que é enviada como resultado para o usuário do nosso sistema, resposta esta que pode ser gerada pelo próprio controlador ou delegada para a camada de visualização. O objetivo deste capítulo é detalhar como este processo ocorre.

Como tudo em Grails, nossos controladores também seguem algumas convenções. Comecemos por aquelas que são aplicadas na criação da classe de um controlador:

- Deve estar armazenada no diretório `grails-app/controllers`.
- Deve possuir o sufixo `Controller` no nome da classe.

Você também pode gerar controladores usando um comando especial presente na interface de linha de comando do Grails chamado `create-controller`, como no exemplo a seguir:

```
create-controller Importacao
```

Serão gerados tanto o arquivo `ImportacaoController.groovy` quanto seu arquivo de teste unitário `ImportacaoControllerSpec.groovy`. Caso não seja fornecido o nome do pacote, será usado como pacote padrão o nome do seu projeto. É importante salientar que você não precisa usar este comando para criar seus controladores. Ele é meramente uma conveniência.

Actions

Observe o método `index` incluído na classe `CategoriaController` pelo scaffolding estático, o qual modifiquei levemente para fins didáticos:

```
def index() {
  [categoriaInstanceList: Categoria.list(params),
   categoriaInstanceCount: Categoria.count()]
}
```

Este método é o que dentro do jargão Grails chamamos de *action*. Neste caso, a única ação executada por nossa primeira action é retornar um mapa contendo duas chaves: `categoriaInstanceList` e `categoriaInstanceCount`, representando respectivamente a lista de categorias e o número de registros presentes no banco de dados.

Mas o que é uma action? É a função que escrevemos em nosso controlador responsável por lidar com as requisições HTTP recebidas pela nossa aplicação web. É seu trabalho tratar os parâmetros recebidos (pela URL, formulários ou corpo da requisição), chamar os métodos necessários na camada de negócio, transformá-los caso necessário para que se tornem adequados à resposta enviada ao usuário do sistema. Esta resposta pode ser essencialmente qualquer coisa: um arquivo (no caso de um relatório ou download), um documento XML, JSON ou em outro formato ou simplesmente uma página HTML.

Você deve ter notado que foi passado como parâmetro à função `list` um objeto chamado `params`. Ele representa os parâmetros que podem ter sido fornecidos ao método através da URL que iniciou sua execução, como neste endereço: http://localhost:8080/concot/categoria?max=10&offset=11, que irá gerar busca que retornará dez itens a partir da décima primeira posição. Falaremos mais sobre o `params` mais à frente. *Mas como esta URL é formada e como é feita a associação entre esta e nossas actions?*

Cuidado com seus métodos!

Fig. 7.5: Atenção redobrada!

Todo método público que você declarar em um controlador será interpretado como uma action. Sendo assim, certifique-se de sempre só incluir métodos nestas classes que atuem como tal.

Essa limitação tem seu lado positivo: ela lhe obriga a incluir sua lógica de negócio em classes de serviço, que são onde este tipo de código realmente deve estar.

Incluir métodos inúteis em seus controladores cria novas URLs que podem inclusive ser usadas por usuários mal intencionados contra seus sistemas. Leve isto em consideração!

Action default

No caso do método `index` estamos lidando com o que no "jargão Grails" chamamos de action padrão (*default action*) do controlador. Quando nossa URL não define qual action deverá ser executada (veremos como isso é feito mais à frente) como Grails sabe qual código executar? No caso, ao acessarmos o endereço http://localhost:8080/concot/categoria Grails irá executar a seguinte lógica:

1) Existe apenas uma action no controlador? Se sim, execute-a.

2) Existe um atributo no meu controlador chamado `defaultAction`? Existindo, este recebe como valor uma string que identifica a action padrão do meu controlador.

3) Existe alguma action chamada "index"? Se sim, será executada.

Voltando ao *ConCot* percebemos que as duas primeiras condições não são satisfeitas: há mais de uma action definida nesta classe e também não temos um atributo chamado `defaultAction`, mas há uma chamada `index`, que será executada.

Executada nossa action, esta retornará um mapa com duas chaves tal como vimos acima e nosso usuário será contemplado com uma página similar à exposta a seguir:

Fig. 7.6: Página index gerada pelo scaffolding

Como esta página é renderizada? Outra convenção do framework: o leitor deve se lembrar que foi gerado um arquivo chamado `index.gsp` no diretório `grails-app/views/categoria`. No diretório `grails-app/views/categoria` encontram-se armazenadas todas as páginas GSP que são usadas **por padrão** pelo controlador. Repare no nome da pasta: corresponde ao do nosso controlador sem o sufixo `Controller`.

E no interior da pasta onde armazenamos nossos arquivos GSP, o que temos? Teremos arquivos cujos nomes corresponderão aos de

nossas actions. Sendo assim, será renderizada por padrão a página `grails-app/views/categoria/index.gsp` para a action `index` presente na classe `CategoriaController`. Simples e direto.

Para que serve aquele mapa que nossa action retornou como valor? Aquele mapa possui um nome especial dentro do "jargão Grails": chama-se **model**, e representa as informações que serão expostas em nossas páginas. Cada uma das chaves definidas nessa estrutura irá corresponder a uma variável que estará disponível em nossos arquivos GSP. Você deve estar curioso para saber como isso ocorre, certo? Então observe o código a seguir que é um trecho do arquivo `index.gsp`.

```
<g:each in="${categoriaInstanceList}" status="i" var="categoriaInstance">
  <tr class="${(i % 2) == 0 ? 'even' : 'odd'}">
  <td>
   <g:link action="show" id="${categoriaInstance.id}">
    ${categoriaInstance.nome}
   </g:link>
  </td>
  </tr>
</g:each>
```

Este é seu primeiro contato com a tecnologia GSP: *Groovy Server Pages*. Falaremos mais a seu respeito adiante, mas por enquanto o que nos interessa neste trecho é o uso que fazemos de um dos atributos do nosso mapa, a chave `categoriaInstanceList`.

A tag `<g:each>` cria um loop responsável por iterarmos sobre todos os itens armazenados em uma coleção. A coleção, no caso, é a chave `categoriaInstanceList` retornada pelo nosso controlador, que iremos passar como valor ao parâmetro `in`. O atributo `status` define o nome de uma variável que atuará como um contador. Nós o usaremos para adicionar a classe *even* ou *odd* a cada uma das linhas de nossa tabela, gerando o efeito listrado que vimos na imagem anterior. Finalmente, o atributo `var` define o nome da variável que será acessada no interior do nosso loop.

A tag `<g:link>` é usada para gerar os links da nossa aplicação a serem renderizados em nossa página. Usamos apenas dois parâmetros: `action`, identificando o nome da action que deverá ser executada em nosso controla-

dor e `id`, cujo valor corresponde ao atributo de mesmo nome do nosso objeto `categoriaInstance`.

Em um arquivo GSP, tudo o que estiver entre `${` e `}` é interpretado e o valor obtido renderizado para o usuário. Soa familiar? Acredito que sim, dado que é exatamente o mesmo mecanismo usado pelas strings em Groovy. :)

Actions no Grails 1.x

Fig. 7.7: Compatibilidade

Uma nota importante para os leitores que estejam vindo do Grails 1.x ou precisem lidar com aplicações implementadas nesta família do framework: a partir da versão 2.0 todas as actions passaram a ser implementadas, por padrão, como funções em nossos controladores, e não mais como closures.

Nas versões anteriores à 2.0 todas as actions eram declaradas desta forma:

```
def index = {
    // código entrava aqui
}
```

Como consequência, o desenvolvedor só poderá trabalhar com binding baseado em mapas e há também o custo oculto no PermGen que uma closure nos trás. Toda closure ao ser compilada é transformada em um novo arquivo `.class`. O resultado é um consumo maior de memória na antiga área PermGen usada pelas versões do Java anteriores à 8.

A mudança trazida pela versão 2.0 nos trouxe alguns benefícios:

- Usa a memória do sistema de uma forma mais eficiente.

- Permite o uso de controladores do tipo singleton (lembre-se de que as closures carregam consigo o escopo de onde são declaradas e fazem, por sua vez, parte do estado do controlador).
- Permite sobrescrever as actions de uma forma mais simples usando herança.
- Permite a interceptação de métodos através de mecanismos de proxiamento padrão, algo que é mais difícil de se fazer com closures, dado que estas acabam sendo confundidas por estes mecanismos com atributos e não métodos.

Agora, se você quiser continuar usando closures em seus projetos (talvez você esteja lidando com código legado) e ter todos os ganhos obtidos com o novo padrão, você pode! Basta adicionar a linha a seguir no arquivo `grails-app/conf/Config.groovy`:

```
grails.compile.artefacts.closures.convert = true
```

Esta instrução irá aplicar uma transformação AST em tempo de compilação no seu projeto, transformando todas as closures declaradas em seus controladores em métodos.

7.3 Dominando URLs

Uma das URLs geradas pela tag `<g:link>` que vimos em nosso arquivo GSP foi http://localhost:8080/concot/categoria/show/1. Há uma série de informações contidas neste endereço que são expostas na imagem a seguir:

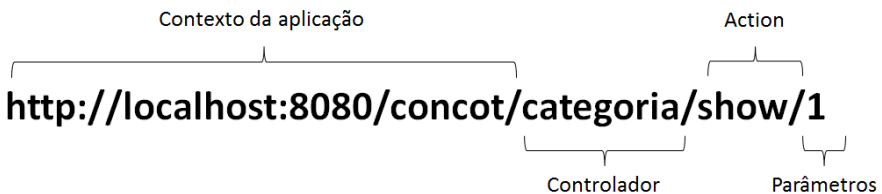



Fig. 7.8: Nossa URL dissecada

O contexto diz respeito à parte da URL que identifica a nossa aplicação dentro de um servidor de aplicações. Qualquer endereço que comece com http://localhost:8080/concot irá cair em alguma action (ou ausência de) do nosso projeto. O que realmente nos interessa aqui é o que vem a seguir: o nome do controlador e qual a action a ser executada.

Se em nosso browser digitarmos esta URL, a action `show` definida na classe `CategoriaController` cujo código-fonte é exposto a seguir (e que modifiquei ligeiramente) será executado.

```
def show(Long id) {
    [categoriaInstance: Categoria.get(id)]
}
```

O leitor deve estar confuso neste momento: é passado o parâmetro `1` à nossa URL, e em nossa action definimos um parâmetro de mesmo nome. Mas onde definimos que aquele " `1`" ao final da URL corresponde ao parâmetro `id` da função `show`?

Novamente outra convenção do Grails: se nossa URL recebe um único parâmetro não nomeado, como é o caso desta URL, ele por padrão recebe o nome de `id`. Este comportamento é inclusive definido via configuração, no arquivo `grails-app/conf/UrlMappings.groovy`, onde configuramos como nossas URLs devem se comportar. O arquivo padrão gerado com toda aplicação Grails é exposto a seguir:

```
class UrlMappings {

    static mappings = {
        // É aqui que definimos a convenção do atributo "id" :)
        "/$controller/$action?/$id?(.${format})?"{
            constraints {
                // apply constraints here
            }
        }

        "/"(view:"/index")
        "500"(view:'/error')
    }
}
```

E como seria a passagem de mais de um parâmetro para uma URL? Voltemos à action `index`, que apenas executa um finder dinâmico. Se quiséssemos expor apenas 10 categorias paginando através da décima primeira posição poderíamos enviar uma URL como a seguinte:

http://localhost:8080/concot/categoria/index?max=10&offset=11

Observe que estamos acessando a action `index` diretamente agora, sem a necessidade de tirarmos proveito das convenções referentes à action padrão do controlador. Apenas para lembrar, a implementação modificada desta action pode ser vista na listagem a seguir:

```
def index() {
  [categoriaInstanceList: Categoria.list(params),
   categoriaInstanceCount: Categoria.count()]
}
```

O objeto `params`, como dito antes, é na realidade um mapa que armazena todos os parâmetros que nossa action recebe através da URL, submissão de formulário ou corpo HTTP. Sendo assim, se modificássemos ligeiramente esta action para imprimir este objeto tal como:

```
def index() {
  println params
  [categoriaInstanceList: Categoria.list(params),
   categoriaInstanceCount: Categoria.count()]
}
```

Antes de a página ser renderizada veríamos no console do Grails ser impresso um texto similar a:

```
[max:10, offset:11]
```

O leitor deve estar confuso agora: por onde são acessados os parâmetros? Pela assinatura do método que compõe a action ou pelo objeto `params`? Pelos dois, só depende da sua preferência. A principal vantagem na passagem dos parâmetros pela assinatura da action está na facilidade de escrita de testes e documentação do que se espera que aquele método receba como entrada. Caso estejamos lidando com muitos parâmetros, ou mesmo situações nas quais os parâmetros recebidos podem variar, sem sombra de dúvidas o

objeto `params` ou `command objects`, que veremos mais à frente irão cumprir muito melhor esta tarefa.

7.4 Redirecionando ações

Em algumas situações, é interessante que uma action redirecione o processamento para outra. Apenas hipoteticamente, imagine que no *ConCot* seja do nosso interesse que um usuário só possa criar novas categorias caso já esteja autenticado no sistema. Basta modificar a action `save` para que fique tal como o código a seguir:

```
def save() {
    if (seguranca.usuarioAutenticado()) {
        // faça o que precisa ser feito
    } else {
        // Redirecione o usuário para a action
        // responsável pelo login
        redirect(action:'login', controller:'entrada')
    }
}
```

Neste exemplo usamos dois parâmetros para identificar qual o controlador e qual action desejamos que seja executada caso o usuário não esteja autenticado. É possível também passar parâmetros para a action a ser redirecionada. Para isso, basta usarmos o parâmetro `params`. Veja o exemplo a seguir no qual incluímos o parâmetro `acessoIndevido` para o controlador `entrada`.

```
def save() {
    if (usuarioAutenticado()) {
        // faça o que precisa ser feito
    } else {
        // Redirecione o usuário para a action
        // responsável pelo login
        redirect(action:'login', controller:'entrada', params:[acessoIndev
    }
}
```

Outra alternativa interessante é redirecionar o usuário para uma URL. Imagine que você esteja desenvolvendo um site voltado a adultos e deseje que menores de idade sejam direcionados para outro site qualquer. Basta usar o parâmetro `url`:

```
def testeIdade() {
    if (params.int('idade') < 21) {
        redirect(url:'http://www.disney.com')
    } else {
        redirect(controller:'home')
    }
}
```

O leitor deve ter notado a função `int` usada no objeto `params`. Assim como `long`, ela converte o parâmetro (que por padrão sempre é armazenado como uma string) para o tipo inteiro ou `long`. No mesmo exemplo, caso o usuário passe no teste da idade, este será então redirecionado para a action padrão do controlador `home`.

7.5 Renderizando a resposta

E se eu não quiser renderizar a página padrão da action, como faço? Usamos a função `render` que se encontra presente em todo controlador.

Imagine que seja do nosso interesse escolher qual página irá renderizar uma categoria recém-persistida de acordo com a permissão de acesso do usuário. Usamos a função `render` presente em todo controlador tal como no exemplo a seguir:

```
def save() {
    /* ignoramos todo o início da action
       o modelo já se encontra armazenado em
       uma variável chamada resultado em nossa action */
    if (condição) {
        render(view:'pagina1', model:resultado)
    } else {
        render(view:'pagina2', model:resultado)
    }
}
```

O método view neste caso recebe dois parâmetros de entrada: `view`, que representa o nome da página GSP que iremos renderizar, por padrão presente no diretório relacionado ao controlador em que a action se encontra; e `model`, representando o modelo a ser usado durante a renderização da mesma.

O arquivo GSP não precisa se encontrar no mesmo diretório do controlador. Se quiser delegar a renderização da resposta para uma página armazenada em outra pasta, basta escrever o caminho completo a ela:

```
def renderizaDiferente() {
    /*
        Renderiza a página contida
        no diretório
        grails-app/views/custom
    */
    render(view:'/custom/pagina')
}
```

Renderizando apenas um trecho HTML, XML ou JSON

Esse exemplo é bastante comum, mas nem sempre queremos renderizar uma página inteira. Há situações, como por exemplo, em uma chamada AJAX, em que nosso controlador deve retornar apenas um trecho HTML. Você pode fazer isso de uma forma bastante simples, como o código adiante, que irá renderizar uma lista completa de categorias embarcadas em HTML.

```
def pesquisa() {
    //executa as consultas necessárias
    //e armazena a lista de categorias em uma
    //coleção chamada resultado
    render(contentType: "text/html") {
        table {
            for (categoria in categorias) {
                tr {
                    td(id:categoria.id) categoria.nome
                }
            }
        }
```

```
    }
}
```

Trata-se de uma DSL simples que irá transformar aquele bloco de código visto anteriormente em HTML similar ao exposto a seguir:

```
<table>
    <tr>
        <td id="1">Equipamentos</td>
        <td id="2">Materiais</td>
    </tr>
</table>
```

Como seria renderizar XML?

```
def pesquisa() {
    // restante omitido
    render(contentType:"text/xml") {
        categorias {
            for (categoria in resultado) {
                categoria(id:categoria.id, nome:categoria.nome)
            }
        }
    }
}
```

Qual o resultado? Algo similar a:

```
<categorias>
    <categoria id="1" nome="Equipamentos"/>
    <categoria id="2" nome="Materiais"/>
</categorias>
```

E JSON, como eu faria? Acho que você já sabe, mas vamos ao exemplo.

```
def pesquisa() {
    // restante omitido
    render(contentType:"application/json") {
        array {
            for (categoria in resultado) {
```

```
                [id:categoria.id, nome:categoria.nome]
            }
        }
    }
}
```

O resultado?

```
[    {id:1, nome:"Equipamentos"},
   {id:2, nome:"Materiais"}  ]
```

Como pode ser visto, é bastante simples: basta passarmos dois parâmetros para o método `render`. O primeiro corresponde ao *mime type* do que desejamos retornar ao cliente da nossa action. O segundo parâmetro é um builder Groovy.

Builders são um poderoso recurso oferecido pelo Groovy que nos permitem representar estruturas hierárquicas como documentos XML, HTML e JSON de uma forma bastante simples. Para maiores informações sobre eles, convém consultar a documentação oficial da linguagem [6].

Interpretado o builder, o método `render` irá simplesmente retornar o resultado diretamente ao cliente que fez a solicitação à nossa action.

Há uma maneira mais simples de se obter o mesmo resultado:

```
def pesquisa() {
    // restante oculto
    render(contentType:"o mime type de sua escolha") {
        objetoQualquer
    }
}
```

Neste caso, o Grails irá percorrer todos os atributos deste objeto gerando o resultado no formato passado como valor para o parâmetro `contentType`. O problema desta abordagem é que você pode gerar documentos enormes acidentalmente caso haja um grafo complexo no que for renderizado. Não considero uma solução tão elegante, portanto. Outro problema é que quando estamos renderizando documentos, na prática estamos criando APIs. Dado que os atributos de nossas classes podem variar, esta variação poderia quebrar todos os clientes que dependam desta API que você acabou de criar.

E ainda é possível simplificar ainda mais esta renderização. Como? Usando o *marshalling* automático de **classes de domínio** usando conversores especiais para os formatos XML e JSON. Trata-se de uma funcionalidade tão fácil de ser usada que nem há muito o que ser dito. Vamos aos exemplos:

```
// Basta incluir esta instrução de importação de classes
import grails.converters.*

// e na sua action...

// Como fazemos com JSON?
def renderizarJSON() {
    // Como renderizamos para JSON uma
    // lista de categorias
    render Categoria.list() as JSON
}

// E para gerar XML?
def renderizarXML() {
    render Categoria.list() as XML
}
```

Simples assim. :)

Marshalling e unmarshalling

Um termo que costuma confundir iniciantes é *marshalling*, possivelmente por não haver uma tradução direta para o português. Marshalling é o ato de estarmos gerando conteúdo em algum formato (XML, JSON) tendo como base um objeto. E o *unmarshalling*? O contrário: você irá instanciar objetos e popular todos os seus atributos tendo como base um documento em um formato específico.

Content negotiation

Muitas vezes você usará Grails para implementar APIs REST (ou SOAP). Se este for o seu caso, você ficará feliz pelo fato de o framework oferecer su-

porte nativo a *content negotiation*. Trata-se de um mecanismo definido dentro do próprio protocolo HTTP que permite o fornecimento de diferentes versões de um mesmo documento representado por uma dada URL.

No caso do Grails, as diferentes versões do documento equivalem aos formatos JSON ou XML. Se neste ponto o leitor mais curioso já brincou um pouco com o scaffolding estático, deve ter percebido algumas actions similares à que exponho a seguir, que representa a listagem de categorias no *ConCot*.

```
def index(Integer max) {
   params.max = Math.min(max ?: 10, 100)
   respond Categoria.list(params), model:[categoriaInstanceCount: Categori
}
```

O que é este `respond`? A função `respond` irá tentar fornecer ao cliente que fez a solicitação a lista de categorias no formato apropriado: JSON, XML ou uma página HTML que é o padrão. No caso, isso pode ser feito através do fornecimento do cabeçalho HTTP `Accept` pelo cliente, pedindo, por exemplo, que a resposta seja enviada no formato XML ou através da inclusão do parâmetro `format` na URL, como http://localhost:8080/concot/categoria?format=xml, que está pedindo que seja fornecido o conteúdo no formato XML.

Por padrão, toda requisição HTTP feita à nossa aplicação costuma vir com o cabeçalho `Accept`, cujo valor default é `*/*`. Para fornecimento no formato JSON, este precisaria vir com o valor `application/JSON` e, para XML, `application/XML`.

A partir do Grails 2.0, você pode inclusive customizar a sua resposta de acordo com o formato pedido pelo cliente:

```
request.withFormat {
    xml {
        // trate o formato XML aqui,
        // exatamente como faria se fosse um render
    }
    json {
        // JSON
    }
    '*' {
        // HTML padrão
```

```
    }
}
```

7.6 Data Binding

Ok, até agora só vimos actions extremamente simples, que não levam em consideração o recebimento de parâmetros. O que ocorre com os dados que submetemos ao controlador usando formulários, query strings em uma URL ou mesmo no corpo de uma requisição? Grails nos fornece um mecanismo de binding extremamente rico e poderoso, que torna sua vida muito mais simples.

O que é binding/data binding? Ao usarmos um framework para desenvolvimento web, muitas vezes nos esquecemos do modo como o protocolo HTTP funciona. Basicamente todos os parâmetros passados às nossas actions vêm como uma estrutura do tipo chave-valor na qual todos os parâmetros encontram-se no formato textual.

Isso trás alguns problemas, dentre os quais podemos citar:

- Como parsear de maneira transparente parâmetros textuais para tipos como `integer`, `long`, `double`, `boolean` etc.?
- Como preencher automaticamente os atributos de um objeto complexo, como uma classe de domínio, a partir daqueles parâmetros?
- Como validar estes parâmetros de tal modo que, estando nossa action esperando um parâmetro do tipo numérico, realmente venha algo condizente com esta condição?
- Preenchimento de formulários: como tornar esta tarefa menos árdua para o programador?

Todos esses problemas são resolvidos por aqueles que se predispõem a escrever frameworks para desenvolvimento web. No caso do Grails, muito do que vemos nesta área é na realidade apenas o reaproveitamento da base adotada pelo framework: o Spring MVC.

Resumindo, o que é *data binding*? É o ato de ligarmos os parâmetros fornecidos ao framework pelo protocolo HTTP às propriedades de um objeto

ou mesmo a um grafo complexo de objetos de forma transparente para o desenvolvedor, que não mais precisa se preocupar com os problemas que citei.

Este procedimento pode ocorrer de diferentes maneiras. Nesta seção serão expostas as mais comuns no dia a dia do programador Grails.

Data binding baseado em mapas

Vejamos como é o formulário de cadastro de categorias gerado pelo scaffolding cujo código modifiquei ligeiramente para facilitar seu entendimento inicial:

```
<g:form action="save" controller="categoria">
  <label for="nome">Nome:</label>
  <br/>
  <g:textField name="nome"/>
  <input type="submit" value="Salvar"/>
</g:form>
```

Há duas novas tags aqui. A primeira é `<g:form>`, que usamos para gerar um formulário cujo atributo `action`, que representa a URL de destino da submissão esteja correta. Ele recebe dois parâmetros que identificam corretamente qual a action a ser executada: `action` e `controller`.

Em seu interior temos outra tag: `<g:textField>`, que irá renderizar um campo textual identificado por "nome". Em seguida, temos um campo do tipo `submit`, que nada mais é do que o botão de submissão deste formulário. O data binding baseado em mapas pode ser feito como na implementação alternativa da action `save` a seguir:

```
def save() {
    // Data binding em ação
    def categoria = new Categoria(params)
    // restante da action pode ser ignorado
}
```

Nesse caso, estamos apenas usando o próprio funcionamento dos POGOs. Lembra que se passarmos para uma classe Groovy um mapa em seu construtor, este irá automaticamente preencher todos os atributos dela? É apenas isso o que fazemos aqui. Se um dos atributos de `Categoria` fosse

do tipo `int`, `boolean`, `long` ou qualquer outro que em Java é primitivo, a própria linguagem se encarregaria de fazer as conversões necessárias para nós.

Há outras maneiras de se usar o objeto `params`, você também pode usá-lo após a instanciação do seu objeto. Isso é muito comum em actions de edição, por exemplo:

```
def update() {
    def categoria = Categoria.get(params.long('id'))
    // Define todos os atributos do nosso objeto em
    // uma única linha
    categoria.properties = params
    // restante da action
}
```

Na realidade, Grails vai um pouco além do mero aproveitamento do comportamento padrão dos POGOs. Imagine que meu objeto `params` seja similar ao exposto a seguir, relativo à persistência de um objeto do tipo `Item`.

```
['categoria.id':'1', nome:'Motor']
```

O atributo `categoria.id` poderia vir, por exemplo, de uma caixa de seleção. Veja a que o scaffolding cria para o cadastro de itens na imagem:

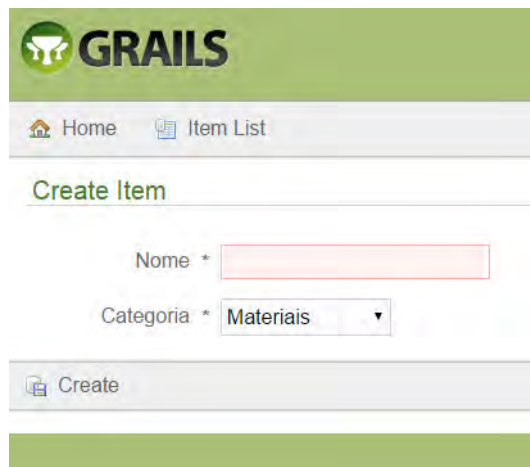



Fig. 7.9: Cadastro de itens

Até o momento apenas vimos o binding com uma entidade de domínio: agora estamos vendo com duas: `Item` e seu atributo do tipo `Categoria`. Grails é inteligente o suficiente para perceber que há um atributo chamado `categoria` em nossa classe `Item` que aponta para outra classe de domínio.

Antes de executar o data binding, Grails obterá uma instância de `Categoria` com o identificador passado como parâmetro e, em seguida, o injetará em nosso objeto. Depois, pode até mesmo alterar algum atributo deste objeto anexado. Imagine que estejamos também modificando o nome da categoria durante a submissão a partir de uma listagem de parâmetros:

```
['categoria.id':'1', 'categoria.nome':'Motores', nome:'Motor']
```

Executando o código a seguir:

```
def save() {
    def item = new Item(params)
    item.save()
}
```

Quando o objeto `Item` for persistido, a categoria cujo identificador é o número `1` terá seu nome alterado para `"Motores"`.

Usando a assinatura da action

Outra maneira bastante elegante de tirarmos proveito do data binding do Grails é através da assinatura da nossa action, ou seja, a partir dos parâmetros que ela espera. Voltando ao exemplo da action `save`, sabe como ela é implementada por padrão pelo scaffolding do Grails? Assim:

```
def save(Categoria categoriaInstance) {
    // conteúdo da action ignorado neste exemplo
}
```

Havendo um único parâmetro na assinatura da action, o data binding será feito diretamente sobre este tal como pôde ser visto. Esse código, na prática, executa o trabalho feito pelo data binding por mapas. A diferença está apenas no modo como escrevemos nosso código.

Uma grande vantagem desta modalidade de binding é que a escrita de testes fica mais fácil e também podemos tirar máximo proveito da tipagem estática, garantindo com isto um desempenho superior para aquele código.

Lidando com erros

Podem ocorrer erros durante o processo de binding? Yeap! Você pode ter erros de tipagem. Imagine que um dos atributos da sua classe de domínio seja do tipo numérico mas seja submetido um valor textual. Novamente, a solução é bastante simples.

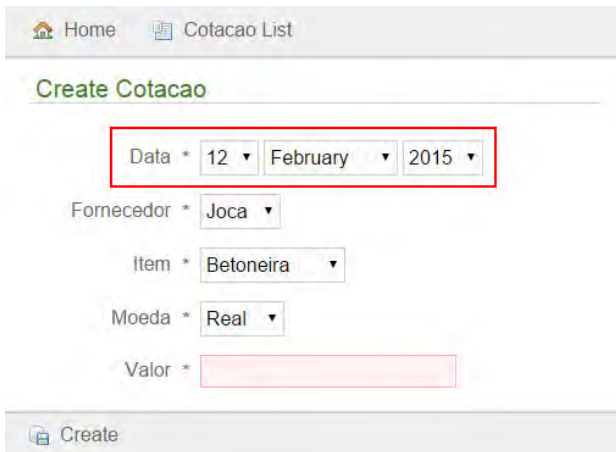
Vamos supor que estamos lidando com a persistência de uma cotação. Apenas para lembrar, esta é a declaração da classe `Cotacao`:

```
class Cotacao {

    BigDecimal valor
    Date data

    static belongsTo = [item:Item,
                        moeda:Moeda,
                        fornecedor:Fornecedor]
}
```

Agora, suponha que estejamos recebendo para fazer o binding os seguintes parâmetros:

```
['valor':'dois reais','moeda.id':'1',
 'item.id':'32', 'fornecedor.id':'3' ]
```

Há um erro claro no parâmetro `valor`. Como resolvemos este problema em nosso controlador?

```
def save(Cotacao cotacao) {
    if (cotacao.hasErrors()) {
        println "Erro no campo ${cotacao.errors.getFieldError('valor')}"
    }
}
```

Simples: o mecanismo de validação entra em ação. Executando a função `hasErrors` da sua classe de domínio, é possível verificar se houve algum erro de validação. Se retornar verdadeiro, já sabe o que pode ter ocorrido.

Usando a função `getFieldError` do atributo `errors` da sua classe de domínio passando como parâmetro o nome do campo é possível checar o que aconteceu. Esta é uma maneira fácil de lidar com esse tipo de situação, com uma vantagem bastante simples: você já iria fazer isto na validação básica das suas classes de domínio.

Mas o leitor deve ter observado que neste exemplo não toquei no binding com campos do tipo `Date`, certo?

Lidando com datas

Há duas maneiras de se lidar com atributos do tipo data. A mais simples consiste em usar a tag `<g:datePicker>`, que renderiza um componente para seleção de data similar ao da imagem a seguir:

Fig. 7.10: Componente date picker

O uso da tag é bastante simples, tal como pode ser visto no formulário a seguir usado no cadastro de cotações:

```
<g:form action="save" controller="cotacao">
```

```
<label for="data">Data:</label>
<g:datePicker name="data" precision="day"/>

</g:form>
```

O atributo `name` irá identificar o nome da propriedade do nosso bean que receberá a data como valor fornecido, enquanto `precision` define o nível de precisão que queremos usar na execução do binding de nossa tag. Os seguintes valores encontram-se disponíveis: `year`, `month`, `day`, `hour`, `minute`, `second`. Quanto menor a unidade de medida selecionada, maior será o número de campos usados para comportar o fornecimento da data.

Claro, não estamos presos à tag `<g:datePicker>`. E quando você quiser usar um campo textual simples, ou mesmo passar uma data como parâmetro de uma URL ou no corpo de uma requisição HTTP? Há dois caminhos a seguir.

O primeiro é simplesmente não fazer nada. Por padrão, o Grails aceita datas que usem os seguintes formatos: `yyyy-MM-dd HH:mm:ss.S` e `yyyy-MM-dd'T'hh:mm:ss'Z`. Se quiser, você pode inclusive incluir os seus próprios formatos de data padronizados. Como fazer isso? Basta adicionar uma chave de configuração no arquivo `grails-app/conf/Config.groovy`: `grails.databinding.dateFormats`, tal como no exemplo abaixo, no qual incluo uma terceira formatação padrão:

```
grails.databinding.dateFormats = ['MMddyyyy',
                                   'yyyy-MM-dd HH:mm:ss.S',
                                   "yyyy-MM-dd'T'hh:mm:ss'Z'"]
```

Sempre que o binding for executado, Grails irá tentar a conversão do campo textual com um dos formatos presentes nesta configuração. A regra de formatação de datas usada é a padrão do Java, definida na classe `java.text.SimpleDateFormat`. Recomendo que você leia o JavaDoc desta classe para compreender melhor seu funcionamento [23].

Mas talvez você queira fazer alguma coisa. Que tal customizar o data binding para um campo específico da sua classe de domínio? Isso é feito com

a anotação `BindingFormat`. Imagine que seja do nosso interesse que os usuários do *ConCot* digitem as datas de cotação no formato `dd/MM/yyyy`. Basta alterar a classe `Cotacao` para que fique desta forma:

```
import org.grails.databinding.BindingFormat

class Cotacao {
    @BindingFormat('dd/MM/yyyy')
    Date data

    // restante omitido
}
```

Customizando a conversão de dados

Aproveitando o gancho do binding de datas, você já se questionou como um texto como `"1/1/2015"` pode ser convertido em um objeto do tipo `java.util.Date` pelo Grails? Conseguimos fazer isso graças à possibilidade de customizarmos o processo de ligação (*binding*).

Para expor esta funcionalidade, vamos implementar o nosso próprio "binder" para datas. Nosso exemplo será apenas um pouco diferente. O formato da data sempre será precedido com o prefixo `"data-"` para se diferenciar do padrão do Grails. A seguir, podemos ver como ficou a implementação da nossa classe:

```
package concot.bind

import org.grails.databinding.converters.FormattedValueConverter
import java.text.SimpleDateFormat

class ExemploFormater implements FormattedValueConverter{
    // Executa a conversão
    def convert(value, String format) {
        def formato = format.replaceAll("data-", "")
        new SimpleDateFormat(format).parse(formato)
    }

    // O tipo de classe que o meu binder irá formatar
```

```
    Class getTargetType() {
        java.util.Date
    }

}
```

A função `convert` recebe dois parâmetros: o valor a ser convertido e a string que identifica o formato a ser aplicado. Seu papel é simplesmente executar a conversão de uma string para um objeto que deve ser do mesmo tipo retornado pela função `getTargetType`.

E como aplicamos isso em nossas classes de domínio ou *command objects* (nosso próximo assunto)? Simples: usando a mesma anotação `@BindingFormat`:

```
class Cotacao {

    @BindingFormat("data-dd/MM/yyyy")
    Date data

}
```

Mas o trabalho ainda não acabou. Há um terceiro passo. Você deve também declarar a nossa classe `ExemploFormater` como um bean do Spring. Falaremos mais sobre isto no capítulo dedicado aos serviços, mas já para adiantar, bastaria modificar o arquivo `grails-app/conf/spring/resources.groovy` para que fique similar ao exemplo a seguir:

```
//Nome do bean (exemploConverter) e classe
exemploConverter concot.bind.ExemploFormater
```

7.7 Command objects

Data binding vai além das suas classes de domínio. Há situações nas quais você irá criar formulários mais complexos que talvez não envolvam qualquer classe de domínio. O que me faz lembrar do dia em que Guto chegou na

DDL pedindo para que fosse incluído um formulário de envio de e-mails para nossos fornecedores...

Sabe o que seria muito legal, Kico? Se no *ConCot* houvesse um formulário no qual eu pudesse enviar um e-mail para os fornecedores.

Como assim, Guto?

Seguinte: eu quero algo mais ou menos assim. Lá no cadastro de fornecedores, uma página que contenha os seguintes campos: o fornecedor, um campo para que eu coloque meu nome e outro no qual eu digite a mensagem. Ao submeter, o formulário enviaria um e-mail para o fornecedor. O que me diz?

Bacana: então vou incluir um atributo a mais na classe `Fornecedor` para representar seu e-mail. E neste formulário de envio de mensagens, falta um campo a mais também Guto, o e-mail para o qual você deseja que o fornecedor envie a resposta, certo?

Perfeito, Kico! Tem como fazer?

Só se for agora!

Nosso primeiro passo é atualizar portanto a classe `Fornecedor` para que fique como o seguinte:

```
class Fornecedor {

    String nome
    String email

    String toString() {
        this.nome
    }

    static constraints = {
        nome nullable:false, blank:false, maxSize:128, unique:true
        email nullable:false, blank:false, email:true
    }
}
```

Repare que neste caso não é necessário salvar a mensagem enviada no banco de dados. Sendo assim, não há razão alguma para criarmos mais uma

classe de domínio. No entanto, seria bacana se houvesse uma maneira interessante de validar os dados que o usuário digita neste formulário. Algumas validações são necessárias:

- Deve ser selecionado um fornecedor;
- O usuário deve fornecer um e-mail válido;
- Alguma mensagem deve ser digitada.

Entra em ação o `command object`. Pense nele como uma "classe de domínio que possui apenas validação". Este tipo de objeto não é persistido, não é buscado em um banco de dados, nada disto: apenas valida a entrada do usuário. E como implementá-lo? Simples, basta criar uma classe Groovy que implemente a *trait* `grails.validation.Validateable`. Criamos uma classe chamada `EnvioEmail`, que se encontra armazenada no diretório `src/groovy/concot` cuja implementação podemos ver a seguir:

```
package concot

class EnvioEmail implements grails.validation.Validateable {

    Fornecedor fornecedor
    String email
    String mensagem

    static constraints = {
        fornecedor nullable:false
        email nullable:false, blank:false, email:true
        mensagem nullable:false, blank:false
    }

}
```

Se excluíssemos a trait `Validateable` desta classe, ela facilmente poderia ser confundida com uma classe de domínio convencional do Grails. É importante salientar também onde salvamos nosso *command object*. Lembre-se que qualquer classe armazenada em `grails-app/domain` irá gerar uma classe de domínio.

Implementado nosso *command object*, o próximo passo é a implementação da nossa primeira action, responsável por expor a página de comunicação:

```
class FornecedorController {

    static scaffold = Fornecedor

    def comunicacao() {
        [fornecedores:Fornecedor.list(), mensagem:new EnvioEmail()]
    }
}
```

O arquivo GSP usado para a renderização (`grails-app/views/fornecedor/comunicacao.gsp`) é exposto em sua forma simplificada adiante:

```
<g:form action="enviarMensagem">
    <label for="fornecedor.id">Fornecedor</label><br/>
    <g:select from="${fornecedores}" name="fornecedor.id" optionKey="id"/>
    <br/>
    <label for="email">E-mail:</label><br/>
    <input type="email" name="email"/>
    <br/>
    <label for="mensagem">Mensagem:</label><br/>
    <textarea name="mensagem">${mensagem.mensagem}</textarea><br/>
    <input type="submit" value="Enviar"/>
</g:form>
```

O resultado final é algo muito parecido com o da imagem:

Fig. 7.11: Nosso formulário

E como fica a action de submissão?

```
class FornecedorController {
    static scaffold = Fornecedor
    // restante omitido
    def enviarMensagem(EnvioEmail envio) {
        envio.validate()
        if (envio.hasErrors()) {
            // Erro encontrado
            flash.message = "Erro de validação"
            render(view:"comunicacao",
                   model:[mensagem:envio,
```

```
                            fornecedores:Fornecedor.list()])
        } else {
            // Mensagem enviada (código omitido)
            flash.message = "Mensagem enviada com sucesso"
            render(view:'comunicacao')
        }
    }
}
```

Todos os erros de validação ficarão armazenados no atributo `errors` do *command object*. Você pode inclusive iterar sobre estes exatamente como faria se fosse uma classe de domínio padrão como vemos no exemplo a seguir:

```
def enviarMensagem(EnvioEmail envio) {
    if (envio.hasErrors()) {
        for (erro in envio.errors.allErrors) {
            println erro
        }
    }
}
```

7.8 Evitando a submissão repetida de formulários

O formulário de comunicação com fornecedores foi um sucesso na *DDL*, de tal modo que acabou se tornando uma das principais ferramentas de comunicação da empresa. Claro, todo sucesso não vem de graça e um dos usuários começou a reclamar que o sistema estava com erros. Alguns fornecedores estavam recebendo duas vezes o MESMO e-mail. Como?

Simples: resultado do famoso "dedinho tenso". Em alguns momentos em que o *ConCot* estava sob forte estresse, alguns usuários acidentalmente clicavam rapidamente duas vezes sobre o botão de submissão. Ainda bem que estamos falando de Grails e a solução do problema é bastante simples: dois passos resolvem o problema. O primeiro passo consiste em usar o atributo `useToken` na tag `<g:form>`:

```
<g:form action="enviarMensagem" useToken="true">
```

```
  <%-- Conteúdo do formulário oculto --%>
</g:form>
```

Este parâmetro inclui um novo parâmetro em nosso formulário, um token, de valor aleatório, que será usado para verificar a submissão repetida de dados. O próximo passo é ainda mais simples: basta modificar levemente nossa action `enviarMensagem`:

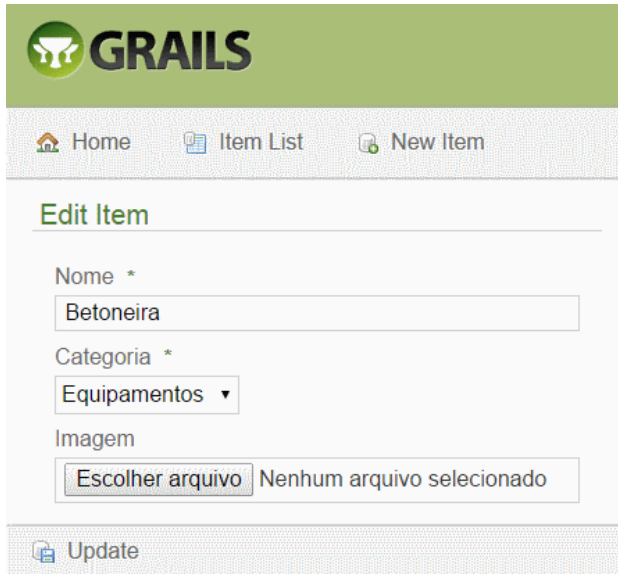
```
def enviarMensagem(EnvioEmail envio) {

    withForm {
        // submissão esperada
    }.invalidToken {
        // submissão duplicada detectada
    }

}
```

Bastou refatorar levemente o código colocando a regra de envio de mensagem dentro do primeiro bloco da função `withForm` e o código que lida com a submissão duplicada dentro do bloco `invalidToken`. Bingo, problema resolvido!

7.9 Upload de arquivos

Conforme o *ConCot* via sua base de dados aumentar, alguns usuários começaram a enfrentar dificuldades na hora de selecionar o item correto de uma cotação. Guto achou que seria uma boa ideia incluir neste cadastro a possibilidade de submissão de arquivos de imagem. Por que não?

A solução mais simples consiste em incluir um atributo do tipo `byte[]` na classe de domínio. Neste caso, iríamos armazenar o arquivo submetido dentro do próprio banco de dados em um campo do tipo BLOB. Para começar, vamos fazer exatamente isso modificando a classe `Item`, desta forma:

```
class Item {

    String nome
```

```
    byte[] imagem

    String toString() {
        this.nome
    }

    static belongsTo = [categoria:Categoria]

    static constraints = {
        nome nullable:false, blank:false, maxSize:128
        categoria nullable:false
        imagem nullable:true, maxSize:65536
    }
}
```

Repare que já incluímos também um limite para o tamanho do arquivo que pode ser submetido: 65536 bytes, ou seja, 64Kb, o que neste caso acreditamos ser um tamanho aceitável. Se você estiver usando scaffolding dinâmico, como estamos fazendo até agora com `Item` não é necessário fazer nada mais. O formulário de cadastro já vai conter o campo para submissão de arquivos. Veja a imagem:

Fig. 7.12: Upload de arquivo

Ao submeter o formulário com um arquivo de imagem, esta automaticamente será armazenada no atributo `imagem` da classe `Item`. O próprio data binding do Grails resolve o problema para nós. Claro, a história não acaba aqui, pois é de nosso interesse "descartar nosso andaime", certo?

O primeiro passo para trabalhar a submissão de formulários é o... formulário. Entra em ação a tag `<g:uploadForm>`, que irá, na realidade, apenas gerar um formulário do tipo `multipart`, que é o usado quando submetemos arquivos no protocolo HTTP. Nosso formulário para o cadastro de itens poderia ser portanto algo similar ao código a seguir:

```
<g:uploadForm action="save">
    <label for="imagem">Imagem</label><br/>
    <input type="file" name="imagem"/>
</g:uploadForm>
```

Resolvemos metade do problema, agora basta ver como obter o arquivo a partir da nossa action. Por baixo dos panos, Grails usa uma interface do

Spring chamada `MultipartHttpServletRequest`, que torna o manuseio de upload uma tarefa trivial. Duvida?

```
class ItemController {
    def save(Item item) {
        def arquivo = request.getFile('imagem')
        if (arquivo.empty) {
            // O arquivo é vazio. Isto é inválido
            // trata o erro
        }
        // armazenamos o arquivo em um sistema de arquivos
        arquivo.transferTo(new File('/arquivos/item/${item.id}'))
    }
}
```

O objeto `request` é uma novidade. Todo controlador possui acesso a este objeto, que na realidade é uma instância de uma conhecida interface da especificação servlet do Java EE: `HttpServletRequest`. A função `getFile` retorna o objeto que representa o arquivo submetido. Mostramos aqui o uso das duas funções mais usadas:

- `empty` retorna `true` caso o arquivo submetido seja vazio ou nulo.
- `transferTo recebe um objeto do tipo` java.io.File

Mas ainda não acabamos...

7.10 Download de arquivos

O download de arquivos pode ocorrer, como sempre, de duas maneiras. A mais simples é aquela na qual não definimos um nome para nosso arquivo: apenas enviamos um array de bytes para o cliente da aplicação através de uma URL.

É uma excelente solução para o caso das imagens que submetemos para nossa classe de domínio `Item`. Entra em ação mais um atributo disponível em todos os controladores: `response`, que na realidade é uma instância de outra famosa interface da especificação Servlet do Java EE, `HttpServletResponse`.

Voltando ao *ConCot*, poderíamos escrever uma action similar à seguinte:

```
class ItemController {
    def imagem(long id) {
        Item item = Item.get(id)
        response.outputStream << item.imagem
    }

    static scaffold = Item
}
```

O operador `<<` foi sobrescrito em todo stream de saída. Do lado direito é fornecido o array de bytes e este será transmitido ao cliente. Em nossos arquivos GSP, obter estas imagens é relativamente simples. Basta fazer como no exemplo a seguir:

```
<img src="${createLinkTo(controller:'item', action:'imagem', id:item.id)}
```

O atributo `src` da tag `<img>` apontará para a nossa action, que atuará como se fosse um acesso direto ao arquivo. Uma solução bastante simples. A função `createLinkTo` na realidade é uma tag presente na biblioteca do Grails. Falaremos mais sobre ela mais tarde. O importante neste ponto é apenas salientar que teríamos uma URL similar a http://localhost:8080/concot/item/imagem/3, por exemplo, que baixaria a imagem para o navegador.

Nossa primeira versão de download foi bastante rudimentar. Normalmente, é interessante fornecer mais alguns atributos do conteúdo baixado para nossos clientes como, mimetype, nome do arquivo e mesmo seu tamanho. Basta chamarmos os métodos da própria interface `HttpServletResponse`.

```
class ItemController {
    def imagem(long id) {
        Item item = Item.get(id)
        // qual o tipo do arquivo baixado?
        response.setContentType('image/png')
        // definimos o nome do arquivo
        response.setHeader('Content-disposition', 'attachment;filename=im
```

```
        // o tamanho do arquivo baixado
        response.setHeader('Content-Length', item.imagem.length)
        // finalmente, o download
        response.outputStream << item.imagem
    }
}
```

É interessante observar que quanto maior for o número de cabeçalhos HTTP fornecidos durante o download, mais fácil se torna o processo de obtenção de dados por parte do cliente. Uma lista completa dos cabeçalhos HTTP pode ser obtida na Wikipédia para consulta [33] (sim, eu sei que Wikipédia não é uma fonte confiável, mas esta lista é bem completa).

7.11 Filtrando requisições

Até este momento, nosso sistema *ConCot* não possui qualquer mecanismo de segurança. Qualquer funcionário da *DDL Engenharia* pode livremente acessar nosso cadastro de cotações. Não seria interessante (para não dizer obrigatório) incluirmos algum mecanismo de autenticação em nosso projeto?

Por enquanto, implementaremos um mecanismo de autenticação extremamente rudimentar (para não dizer ineficiente). Nosso objetivo não é tratar da segurança de nossas aplicações Grails neste capítulo, mas sim apresentar mais alguns conceitos fundamentais relacionados ao funcionamento dos controladores.

Este nosso mecanismo de autenticação terá três componentes: uma nova entidade representando nossos usuários, um formulário de autenticação e um mecanismo de filtragem de requisições. O usuário que fornecer as credenciais corretas irá ter acesso a todos os dados, pois não iremos implementar aqui um mecanismo de autorização. Nosso objetivo neste momento é apenas apresentar o mecanismo de filtragem de requisições do Grails.

A classe de domínio `Usuario` é bastante simples:

O "boot" do Grails

```
class Usuario {
```

```
    String login
    String senha

    static constraints = {
        login nullable:false, blank:false, maxSize:16
        senha nullable:false, blank:false, maxSize:312
    }
}
```

Repare que quando disse que nosso mecanismo de autenticação era simples eu não estava mentindo, pois nós sequer iremos salvar a senha de nossos usuários encriptadas no banco de dados. Como todo sistema precisa ter pelo menos um usuário, vamos criá-lo no momento em que a aplicação é iniciada. Para tal, usamos o arquivo `grails-app/conf/BootStrap.groovy` que, após nossa modificação, ficou assim:

```
import concot.*

class BootStrap {

    def init = { servletContext ->
        Usuario.findOrSaveByLoginAndSenha("admin", "senha")
    }

    def destroy = {
    }
}
```

No bloco `init` incluímos as instruções que desejamos que sejam executadas quando nossa aplicação é iniciada pelo servlet container. É o local ideal para criarmos nosso usuário administrador padrão, razão pela qual o criamos ali usando nosso finder dinâmico.

Ah, e aquele bloco `destroy`? Ele é executado quando nossa aplicação é finalizada pelo container. Um bloco bastante útil quando queremos limpar qualquer bagunça que tenha sido gerada por nossa aplicação durante sua execução.

Sessão de usuário e nossa autenticação

Agora que já temos pelo menos o primeiro usuário cadastrado quando o sistema é iniciado, nosso próximo passo consistirá em implementar o mecanismo de autenticação. Para isso, primeiro iremos criar um novo controlador chamado `Autenticacao`, cuja implementação é bem simples:

```
class AutenticacaoController {

    def autenticar(Usuario usuario) {
        def registro = Usuario.findByLoginAndSenha(usuario.login, usuario.s
        if (registro) {
            session['usuario'] = registro
            redirect(controller:'cotacao')
        } else {
            flash.message = "Acesso negado"
            redirect(uri:'/')
        }
    }

}
```

O funcionamento é extremamente simples. Verificamos se existe um registro de usuário com o login e senha passados como parâmetro ao controlador. Existindo, redirecionamos nosso cliente para a action default do controlador `cotacao`; não havendo, o redirecionamento será feito para a página inicial do projeto.

A primeira novidade diz respeito ao modo como lidamos com a sessão do usuário. Todo controlador possui um atributo chamado `session`, que funciona como um mapa. Os objetos que inserirmos em seu interior irão lá permanecer durante o tempo de vida da sessão do usuário. No nosso caso, apenas incluímos aqui a instância de `Usuario` que encontramos no banco de dados.

Armazenar o usuário na sessão é algo fundamental em nosso mecanismo de segurança como veremos mais à frente. A segunda novidade é o escopo `flash`. Ele consiste em um conjunto de variáveis de duração extremamente curta: ele é capaz de manter os valores armazenados em si apenas na requisição corrente e na próxima, o que é ideal para o nosso caso, pois queremos

informar à pessoa que tentou se autenticar no sistema que seu acesso foi negado.

Precisamos também modificar a página inicial do *ConCot*. Esta agora possui apenas um formulário de autenticação cuja implementação pode ser vista a seguir:

```
<h1>ConCot</h1>
<g:form action="autenticar" controller="autenticacao">
    Login:<br/>
    <input type="text" name="login"/><br/>
    Senha:<br/>
    <input type="password" name="senha"/><br/>
    <input type="submit" value="Entrar"/>
</g:form>
```

Nossa página inicial possui agora um aspecto similar ao da imagem:

Fig. 7.13: Formulário de autenticação

Finalmente, o filtro

O último componente do nosso dispositivo de segurança é o filtro. Filtros são classes que nos permitem interceptar todas as requisições que chegam aos

nossos controladores. Novamente, devem ser seguidas algumas convenções para que possamos implementar nossos filtros.

- A classe deve ser criada no diretório `grails-app/conf`;
- A classe deve possuir o sufixo `Filters` em seu nome.

O filtro de acesso do *ConCot* encontra-se implementado no arquivo `grails-app/conf/concot/AcessoFilters.groovy`. Vamos escrevê-lo juntos para que você entenda o funcionamento deste poderoso recurso oferecido pelo Grails. A primeira versão do nosso filtro pode ser vista na listagem adiante:

```
package concot

class AcessoFilters {
    def filters = {

    }
}
```

Deve ser declarado um bloco `filters` em toda classe de filtro. Caso esteja ausente, nossa aplicação simplesmente não será iniciada. Em seu interior é que declaramos as regras de interceptação de requisições. Em nosso caso há apenas uma:

```
class AcessoFilters {
    def filters = {
        acesso(controller:'*', action:'*') {
            // preencheremos aqui mais tarde
        }
    }
}
```

Nossa primeira regra se chama `acesso` e os parâmetros que declaramos em seu interior nos dizem o que deve ser interceptado. Neste caso, estamos interceptando todas as actions pertencentes a qualquer controlador, razão pela qual para os dois parâmetros passamos como valor um asterisco. Mas você pode ser muito mais específico. Veja alguns exemplos:

```
// Todas as actions do controlador cotacao
acesso(controller:'cotacao', action:'*')
// Apenas a action save do controlador cotacao
acesso(controller:'cotacao', action:'save')
// Talvez todas as actions save de qualquer controlador
acesso(controller:'*', action:'save')
```

Mas não temos apenas os parâmetros `controller` e `action`. Segue a lista de todos os operadores que podemos usar para filtrar nossas requisições:

- `controller` o nome do controlador. Por padrão seu valor é '*' caso não o inclua na definição.
- `controllerExclude` no caso de uma regra que se aplique a todos os controladores menos algum. Exemplo: `controllerExclude:'item'`. Todos os controladores menos item.
- `action` o nome da action a ser interceptada. Seu valor padrão é '*' caso não o forneça em sua regra.
- `actionExclude` o mesmo que `controllerExclude`, só que relacionado a actions.
- `regex` aplicado a URLs.
- `uri` Aplicado a um endereço. Por exemplo: `uri:/item/**`.
- `uriExclude` O mesmo que `controllerExclude`, só que aplicado a URIs.
- `invert` Inverte a sua definição de interceptação.

Mas não basta saber o que interceptar: você também precisa saber **quando**. Por esta razão Grails nos fornece três tipos de filtros:

- `before` de longe o mais comum, no qual você intercepta a requisição antes que esta execute a sua action.

- `after` após a execução da action
- `afterView` após a renderização da resposta

Em nosso caso, iremos implementar um filtro do tipo `before`, sendo assim nossa implementação ficará desta forma:

```
class AcessoFilters {

    def filters = {
        acesso(controller:'*', action:'*') {
            before = {

            }
        }
    }

}
```

Mas você também pode incluir dentro de uma mesma regra mais de um tipo de filtro, tal como no exemplo a seguir:

```
class AcessoFilters {
    def filters = {
        acesso(controller:'*', action:'*') {
            before = {

            }
            after = {

            }
            afterView = {

            }
        }
    }
}
```

Bom, mas como ficou nosso filtro no final das contas?

```
class AcessoFilters {

    def filters = {
        acesso(controller:'*', action:'*') {
            before = {
               if (session['usuario']) {
                       return true
               } else {
                       if (controllerName == null || actionName == 'auten
                           return true
                       }
                       redirect(uri:'/')
               }
            }
        }
    }

}
```

Se o filtro retornar o valor `true`, a action será executada como se nada tivesse ocorrido. Nossa regra de acesso, como pôde ser visto, é bastante espartana: se o usuário estiver autenticado (houver algo na chave `usuario` da sessão), todo o acesso é liberado.

O realmente interessante ocorre quando o usuário não está autenticado no sistema. Há uma série de variáveis que se encontramos disponíveis para o nosso filtro, e nós usamos algumas. A variável `controllerName` indica o nome do controlador da requisição corrente. No caso da página inicial do sistema, não há nenhum, por isto retornamos `true` (há de existir um ponto de entrada no sistema afinal de contas).

Já a segunda variável disponibilizada é `actionName`, que identifica o nome da action. A única action livre em nosso sistema é `autenticar`, razão pela qual iremos retornar `true` também nesta condição. Finalmente, não passando por este rápido teste, simplesmente redirecionamos o nosso visitante para a página inicial do sistema caso tente acessar alguma URL do sistema a que não possua acesso.

Sobre as variáveis disponíveis para o filtro, segue a lista:

- `request` o objeto `HttpServletRequest` da API Servlet
- `response` o objeto `HttpServletResponse` da API Servlet
- `session` a sessão do usuário
- `servletContext` o objeto `ServletContext` da API Servlet
- `flash` o contexto flash
- `actionName` o nome da action interceptada
- `controllerName` o nome do controlador interceptado
- `applicationContext` o objeto `ApplicationContext`

Você também tem acesso a dois métodos importantes:

- `redirect` para fazer os redirecionamentos necessários.
- `render` para renderizar respostas customizadas em seus filtros.

Mas ainda podemos filtrar ainda mais

Neste momento, talvez você ache que nosso trabalho está completo, mas verdade seja dita, ainda não está. Falta um pequeno detalhe. Imagine que alguém tente acessar a seguinte URL: http://localhost:8080/concot/autenticacao/autenticar?login=admin&senha=senha. A pessoa conseguirá se autenticar no sistema, e ainda vemos a senha na barra de endereços do navegador. Como resolver isto? Limitando quais métodos HTTP possuem acesso às nossas actions!

Entra em ação o atributo `allowedMethods` em nosso controlador. Vamos incluí-lo em nossa classe `AutenticacaoController` para que fique similar ao código a seguir:

```
class AutenticacaoController {
    static allowedMethods = [autenticar:'POST']
}
```

Agora as credenciais dos nossos usuários serão fornecidas apenas através do método POST HTTP, ou seja, através de formulários apenas. Problema resolvido. Se você quiser, é possível definir mais de um método HTTP para uma mesma action. Basta que você inclua como valor a esta uma lista:

```
static allowedMethods = [autenticar:'POST', sair:['GET','DELETE']]
```

7.12 Escopo do controlador

O controlador Grails na realidade é um bean do Spring. Veremos mais sobre beans e escopos quando falarmos sobre serviços 9.2, porém é importante que você entenda o essencial deste conceito agora com o objetivo de otimizar o consumo de recursos computacionais do seu projeto e também evitar alguns problemas.

Quando falamos de escopo, estamos nos referindo, essencialmente, ao tempo de vida de uma instância. Por quanto tempo deverá existir a instância de um objeto do tipo CotacaoController, por exemplo? O tempo de uma requisição? A sessão de um usuário? Todo o tempo de vida da aplicação?

No caso dos controladores, há suporte para três tipos de escopo:

- prototype é o padrão caso a aplicação não seja configurada para um escopo diferente. Uma nova instância do controlador será criada a cada requisição e finalizada quando seu processamento chegar ao fim.
- session a instância do controlador existirá enquanto durar a sessão do usuário.
- singleton a instância é criada junto com a inicialização da aplicação e será mantida enquanto esta estiver online.

Se você quiser alterar o escopo padrão de todos os controladores da sua aplicação, basta incluir a chave grails.controllers.defaultScope no arquivo grails-app/conf/Config.groovy:

```
//Mudando o padrão para session
grails.controllers.defaultScope='session'
```

É importante observar que na configuração padrão do arquivo `Config.groovy` a chave `grails.controllers.defaultScope` já está vindo há algum tempo com o valor `singleton`.

Também é possível mudar o escopo de apenas um controlador. Para tal, basta incluir o atributo estático `scope` em sua definição:

```
class ItemController {
    // Mudando o escopo para prototype
    static scope = "prototype"

}
```

Cuidado com o estado

Fig. 7.14: Atenção redobrada!

O escopo *singleton* é o ideal se seu objetivo for minimizar o consumo de memória do sistema, visto que será mantida uma única instância do seu controlador durante todo o tempo de vida da aplicação. Entretanto, muito cuidado deve ser levado na implementação dos seus controladores.

Evite ao máximo a presença de estado em seus controladores ao adotar este escopo. *Que estado?* Simples, atributos em seu controlador que não sejam actions, e que possam ser alterados de acordo com as requisições que este possa receber.

Preferencialmente, implemente seus controladores sempre sem qualquer atributo que não seja um serviço 9: idealmente controladores deveriam possuir apenas actions.

Caso seja necessário incluir estado em seus controladores, opte pelos escopos `scope` ou `session`, evitando assim que um usuário acidentalmente obtenha acesso ao estado de outro. Lembre-se: aplicações web são essencialmente aplicações concorrentes!

7.13 Escopo de dados

Assim como há escopo para controladores, há também escopo para dados. Vimos alguns neste capítulo: `flash`, `session` e `params`. Tratam-se de estruturas do tipo mapa que usamos para armazenar algumas variáveis fornecidas pelo usuário em suas requisições ou definidas pelo programador em seus controladores ou filtros.

Sua principal função é definir o tempo de vida de variáveis. Você já sabe como trabalhar com estas estruturas, dado que vimos isso o capítulo inteiro. O que talvez não conheça são as definições formais de todos os escopos disponíveis ao programador:

- `params` os parâmetros enviados ao controller através das requisições.
- `session` a sessão do usuário.
- `flash` um escopo que armazena as variáveis somente durante a requisição corrente e a próxima para um mesmo usuário.
- `servletContext` variáveis que se encontram disponíveis durante todo o tempo de vida da aplicação. Use com cuidado este escopo, pois este acaba se tornando uma espécie de "variável global" dentro do seu projeto web.
- `request` armazena variáveis apenas durante o tempo de vida da requisição corrente.

Fig. 7.15: Atenção redobrada!

Ao lidarmos com escopos, o leitor deve sempre se lembrar do que disse agora a pouco: **uma aplicação web é essencialmente uma aplicação concorrente**. Você terá o tempo inteiro mais de um usuário acessando uma mesma

URL ou recurso ao mesmo tempo. Sendo assim, tenha **máximo cuidado** com o escopo `servletContext`. Só inclua valores neste escopo que sejam imutáveis durante o tempo de vida da aplicação, dado que são informações compartilhadas por todos os usuários do sistema.

Capítulo 8

A camada web: visualização

GSP (*Groovy Server Pages*) é **A** camada de visualização oferecida pelo Grails. Na documentação oficial é dito que foi feita pensando programadores familiarizados com tecnologias como JSP e ASP, mas acredito que alguém acostumado com PHP se sentirá em casa com o que mostrarei aqui. Digo isso por experiência própria, pois já constatei que programadores PHP costumam gostar bastante do Grails. Espero que, caso você venha do PHP, este seja seu caso também. :)

Se você já programou em JSP, vai se sentir bastante aliviado neste primeiro contato com GSP, já que ele nos livra de uma série de burocracias a que este nos força, por exemplo, a necessidade de declararmos nossas tags, ou mesmo de usarmos nossas tags apenas como se fossem tags, como veremos neste capítulo.

8.1 O essencial

Você cria seus arquivos GSP exatamente como faria se estivesse trabalhando com uma página HTML convencional, o que torna a tecnologia uma excelente alternativa para os membros da equipe que não trabalham diretamente com programação. O arquivo a seguir é um GSP perfeitamente válido:

```
<!DOCTYPE html>
<html>
    <head>
        <title>GSP ou HTML?</title>
    </head>
    <body>
        <h1>Apenas o HTML que você já conhece</h1>
        <p>Designers adoram!</p>
    </body>
</html>
```

Não é preciso declarar bibliotecas de tag (*tag libraries*) ou incluir diretivas como no caso do JSP, no qual esta é uma tarefa praticamente obrigatória na maior parte das vezes. Como vimos no capítulo sobre controladores 7, a escolha do GSP a ser renderizado pode se dar tanto por convenções quanto pela definição explícita do arquivo pelo controlador. O interessante é você ter em mente que **jamais** esses arquivos serão renderizados diretamente por uma URL, apenas através de controladores, bibliotecas de tag ou templates (de que falaremos a respeito mais à frente).

Scriptlets

A tecnologia começa a ficar interessante justamente quando falamos sobre o recurso cujo uso é o mais desencorajado: `scriptlets`! Scriptlets são, como o próprio nome já diz, pequenos scripts (preferencialmente) que você pode embutir em seus arquivos GSP. Idealmente, suas páginas devem ser simples o suficiente de tal modo que este recurso não seja usado, mas caso seja realmente inevitável... use-o com extrema moderação.

Declarar um scriptlet é simples e bastante familiar a programadores JSP: basta escrevê-los entre `<%` e `%>` tal como no exemplo a seguir referente ao

arquivo `scriptlet.gsp`. O que digitamos em seu interior? Código Groovy, é claro!

```
<!DOCTYPE html>
<html>
    <head>
        <title>GSP ou HTML?</title>
    </head>
    <body>
        <h1>Apenas o HTML que você já conhece</h1>
        <p>Designers adoram!</p>
        <%
        // Declaro uma variável chamada "data" aqui
        def data = new Date()
        %>
        <p>A propósito, agora é <%= data %></p>
    </body>
</html>
```

Claro, a página não pode ser acessada diretamente, sendo assim criei um controlador apenas para testar o recurso, que se chama `VisualizacaoController`.

```
class VisualizacaoController {
    def scriptlet() { }
}
```

E ao acessar a URL http://localhost:8080/concot/visualizacao/scriptlet o que obtemos?

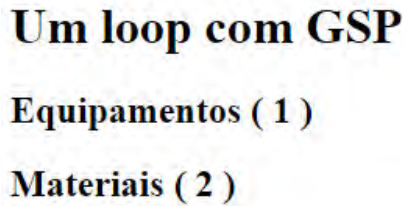
Apenas o HTML que você já conhece

Designers adoram!

A propósito, agora é Sun Feb 15 10:53:25 BRST 2015

Fig. 8.1: Scriptlets!

Seu scriptlet, claro, pode ser muito mais complexo que o deste exemplo, falaremos sobre isto daqui a pouco, mas o interessante neste momento é o trecho que saliento a seguir:

```
<p>A propósito, agora é <%= data %></p>
```

Entre `<%=` e `%>`, é incluída uma expressão qualquer (que pode ser também uma variável, como fiz neste exemplo). O resultado será embutido na renderização final da página, mas há uma maneira mais interessante de se obter o mesmo resultado:

```
<!DOCTYPE html>
<%
  // Declaro uma variável chamada data, aqui
  def data = new Date()
%>
<%-- Que tal uma expressão Groovy? --%>
<p>A propósito, agora é ${data}</p>
```

Basta incluir sua expressão Groovy entre `${` e `}`. Esta é inclusive a maneira recomendada de se incluir expressões em seus GSP, pois é muito mais simples e lhe afastará (ao menos em teoria) dos scriptlets. Você também pode implementar loops ou condicionais com GSP de uma forma muito parecida com o que fazemos em JSP ou PHP:

```
<h1>Um loop com GSP</h1>
<%
  def lista = concot.Categoria.list()
  for (item in lista) {
%>
<h2>${item.nome} ( ${item.id} )</h2>
<%
}
%>
```

Cujo resultado durante a renderização pode ser visto a seguir:

Um loop com GSP

Equipamentos (1)

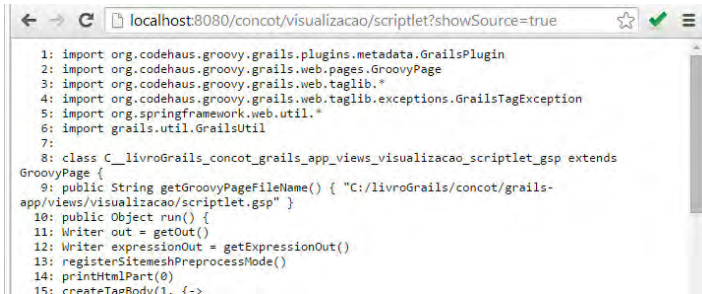
Materiais (2)

Fig. 8.2: Aplicando loops

Claro, código similar também poderia ser feito usando condicionais sem problema algum:

```
<h1>Um loop com GSP</h1>
<%
  def lista = concot.Categoria.list()
  for (item in lista) {
    /* Renderizaria apenas aqueles com id
     maior que 1 */
    if (item.id > 1) {
%>
<h2>${item.nome} ( ${item.id} )</h2>
<%
    }
}
%>
```

Comentários também podem ser incluídos em arquivos GSP. Estes funcionam exatamente como faríamos em arquivos JSP, ou seja, estarão visíveis apenas no código-fonte, e não durante a renderização. Como fazemos isto?

```
<%-- Comentário com uma linha --%>
<h1>Um loop com GSP</h1>
<%
  def lista = concot.Categoria.list()
  for (item in lista) {
%>
<h2>${item.nome} ( ${item.id} )</h2>
<%
}
%>

<%--
    Um comentário
    com mais de uma
    linha.
--%>
```

Tudo o que estiver entre `<%--` e `--%>` será ignorado pelo compilador GSP em tempo de execução.

Depurando GSP

A propósito, arquivos GSP, assim como JSP costumam oferecer desempenho superior pelo fato de serem compilados antes de serem executados. Haverá momentos (torço para que os seus sejam extremamente raros), conforme suas páginas se tornam mais complexas, que você terá dificuldade em entender o que realmente está ocorrendo em seu interior ou mesmo impedindo um desempenho excelente. Um recurso bastante interessante no GSP é a possibilidade de podermos visualizar, apenas no ambiente de desenvolvimento, o código-fonte que é gerado pelo compilador.

Como faço isso?. Simples, apenas adicione o parâmetro `showSource=true` ao final da sua URL, tal como http://localhost:8080/concot/visualizacao/scriptlet?showSource=true. O resultado é a renderização do código-fonte na janela do seu navegador como pode ser visto a

seguir:

Fig. 8.3: Usando o parâmetro showSource

8.2 Tags customizadas

Tags customizadas são uma solução de componentização para a camada de visualização. Como veremos nesta seção, o modo como GSP lida com este recurso é tão simples quanto a sintaxe GSP que acabamos de ver e muito mais fácil que o modo como estamos acostumados a trabalhar com JSP: não é preciso incluir declarações de importação no topo da sua página, tudo o que você precisa fazer é usá-las.

GSP já vem com uma biblioteca de tags padrão cujo prefixo é `g:`. Acredito que a melhor maneira de apresentar as tags customizadas a você é primeiro apresentar as fundamentais para, logo em seguida, implementar as nossas próprias *tag libraries*.

Renderização condicional

Tags condicionais definem o que deverá ser renderizado ao cliente de acordo com uma condição booleana. GSP nos fornece algumas tags para isso, sendo a mais usada sem sombra de dúvidas `<g:if>` tal como vemos no exemplo a seguir:

```
<g:if test="${1 + 2 < 4}">
<p>Sim, 1 + 2 é 3, que é menor que 4</p>
</g:if>
```

O atributo `test` recebe como parâmetro uma expressão Groovy que, se retornar verdadeiro, irá renderizar seu conteúdo embarcado (também chamado de "corpo", o *body* da tag). Mas este não é o único atributo aceito por `<g:if>`: outro bastante útil durante desenvolvimento é `env`, que recebe como parâmetro o nome do seu *environment*, o que possibilita, por exemplo, expor informações de depuração apenas no ambiente de desenvolvimento como no exemplo a seguir:

```
<g:if env="development">
<p>Informações úteis ao desenvolvedor apenas</p>
</g:if>
```

Sabem para que o atributo `env` também é bastante prático? Para a inclusão de código de marcação de terceiros que só seja interessante ser renderizado em produção como, por exemplo, contadores ou anúncios em sua página.

```
<g:if env="production">
<%-- Código do seu contador de acessos --%>
</g:if>
```

Temos também a tag `<g:else>` que nos permite renderizar conteúdo caso determinada condição *não* tenha sido atendida. Esta sempre deve ser incluída logo após `</g:if>` ou `</g:elseif>`.

```
<g:if test="${1 + 2 < 2}">
<p>A matemática enlouqueceu!</p>
</g:if>
<g:else>
<%-- Este bloco será renderizado --%>
<p>Ainda há sanidade matemática</p>
</g:else>
```

Também podemos contar com `<g:elseif>`, que funciona exatamente como `<g:if>`: a única diferença está no fato de que nos permite tratar um número maior de casos durante a renderização condicional.

```
<g:if env="production">
  <p>Ambiente de produção</p>
```

```
</g:if>
<g:elseif env="development">
  <p>Ambiente de desenvolvimento</p>
</g:elseif>
<g:else>
  <p>Nem produção nem desenvolvimento, mas sim
  ${grails.util.GrailsUtil.getEnvironment()}</p>
</g:else>
```

O primeiro bloco que atender à condição será o renderizado. A classe `grails.util.GrailsUtil` faz parte da API padrão do Grails e foi usada para nos retornar em tempo de execução qual o *environment* corrente da aplicação.

Iterações

Em diversos momentos, é necessário que você itere sobre uma coleção de itens ou execute algum loop enquanto determinada condição se mostrar verdadeira. Lembra da página de listagem gerada pelo *scaffolding* do Grails?

Fig. 8.4: Listagem de categorias

Ela é escrita usando a tag `<g:each>`, tal como no exemplo a seguir:

```
<g:each in="${categoriaInstanceList}" status="i" var="categoria">
 <tr class="${(i % 2) == 0 ? 'par' : 'impar'}">
   <td>
     <g:link action="show" id="${categoria.id}">${categoria.nome}</g:link>
   </td>
 </tr>
</g:each>
```

O corpo da tag será renderizado enquanto ainda houver itens a serem percorridos na coleção que passamos como valor ao parâmetro `in` (parâmetro obrigatório). Já o parâmetro `var` define o nome da variável que representa o item corrente em nossa iteração. Outro atributo bastante útil é `status`, que define o nome da variável que armazenará a posição corrente do item dentro da iteração e que pode ser usado para, por exemplo, construir o efeito listrado usado na renderização da nossa listagem.

Outra tag que pode ser usada para lidar com iterações é `<g:while>`, que possui um único parâmetro: `test` que, se verdadeiro, irá renderizar o corpo da tag. A seguir, mostro como expor uma contagem usando este recurso:

```
<% corrente = 0 %>
<g:while test="${corrente < 10}">
  <p>O valor corrente é ${corrente}</p>
  <% corrente++ %>
</g:while>
```

Fig. 8.5: Atenção redobrada!

Muito cuidado com `<g:while>`: ao escrever seu código, certifique-se de que a condição de finalização do loop chegará a um valor falso em algum momento. Ignorar esse fato poderá gerar um loop infinito que possivelmente consumirá todos os recursos computacionais do seu servidor.

Definição de variáveis

Já definimos algumas variáveis em nossas páginas antes, mas como lhe recomendei no início deste capítulo, devemos evitar o uso de scriptlets. A equipe responsável pelo desenvolvimento do Grails concorda, razão pela qual criou a tag `<g:set>`.

Esta tag possui dois parâmetros obrigatórios: `var`, que define o nome da variável, e `value`, que definirá o seu valor. Nosso último exemplo pode ser facilmente reescrito usando esta tag:

```
<g:set var="corrente" value="${0}"/>
<g:while test="${corrente < 10}">
  <p>O valor corrente é ${corrente}</p>
  <g:set var="corrente" value="${corrente++}"/>
</g:while>
```

Outro parâmetro interessante presente nesta tag é `scope`, que nos permite armazenar a variável em qualquer um dos escopos de dados 7.12 que vimos no capítulo sobre controladores. Sendo assim, se você quiser armazenar algum valor na sessão do usuário basta usar a tag tal como no exemplo a seguir:

```
<g:set var="autorDesteLivro"
       value="Henrique Lobo Weissmann"
       scope="session"/>
```

Links

Uma tag que você com certeza viu sendo usada no capítulo sobre controladores 7 foi `<g:link>`, que renderiza na sua página uma tag `<a>` contendo a URL correta que apontará para sua action alvo. A melhor maneira de apresentá-la é usando-a, sendo assim, vamos a alguns exemplos:

```
<%--
Para gerar...
<a href="http://localhost:8080/concot/cotacao">Cotações</a>
--%>
<g:link controller="cotacao">Cotações</g:link>
```

```
<%--
Para gerar
<a href="http://localhost:8080/concot/cotacao/create">Criar</a>
--%>
<g:link controller="cotacao" action="create">Criar</g:link>
```

Fig. 8.6: Atenção redobrada!

Lembre-se que o contexto da sua aplicação irá variar de acordo com o seu environment. Sendo assim, sempre que for criar links internos use a tag `<g:link>` (ou alguma das variantes expostas nesta seção), pois isto garante que serão gerados links com a URL correta para você.

Você também pode incluir na tag `<g:link>` os parâmetros necessários para montar sua URL. Para isso, podemos usar dois parâmetros:

- `id` Quando você quer passar apenas o atributo `id` à sua URL, como por exemplo em http://localhost:8080/categoria/show/1.
- `params` Recebe como parâmetro um mapa usado para a composição do seu link.

Vamos a alguns exemplos:

```
<%--
Para gerar
<a href="http://localhost:8080/cotacao/show/1">Expor</a>
--%>
<g:link controller="cotacao" action="show" id="${cotacao.id}">Expor</g:lin
<%--
Para gerar
<a href="http://localhost:8080/cotacao?max=10&offset=11">Listagem</a>
--%>
<g:link controller="cotacao" params="[max:10,offset:11]"/>
```

Talvez você não precise da tag `<a>`, mas apenas da URL gerada. Nesse caso, usamos a tag `<g:createLink>`.

createLinkTo

Fig. 8.7: Atenção redobrada!

Há uma tag chamada `<g:createLinkTo>` no Grails que foi marcada como obsoleta (*deprecated*). Cuidado para não confundir com `<g:createLink>`. Evite-a.

Ela é usada quando precisamos incluir um link em algum trecho do nosso GSP, como em instruções JavaScript nas quais só precisamos do endereço. Podemos ver um exemplo a seguir:

```
<%-- Um exemplo usando jQuery --%>

<script type="text/javascript">
jQuery.get("${createLink(controller:'controller', action:'show', id:1)}",
    function(data) {
        // nosso código aqui
    })
</script>
```

Reparou que usamos a tag `<g:createLink>` como se fosse uma função? Sim: podemos fazer isto com GSP. Tags podem ser chamadas tanto no seu estado natural quanto como funções. Falaremos mais sobre isso mais à frente neste capítulo 8.2.

Muitas vezes você irá querer referenciar conteúdo estático. Imagine que tenhamos um arquivo CSS que seja de nosso interesse incluir em nossa pá-

gina. Entra em ação a tag `<g:resource>`, que podemos usar tal como no exemplo a seguir:

```
<head>
  <link rel="stylesheet" href="${resource(dir:'css', file:'main.css')}"/>
</head>
```

Neste exemplo, referenciaremos o arquivo estático `web-app/css/main.css`. Yeap: podemos armazenar conteúdo estático no diretório `web-app` da nossa aplicação. O atributo `dir` da tag apontará para o diretório relativo em seu interior e `file`, o arquivo que queremos incluir em nosso link. Também falaremos mais sobre recursos estáticos 8.8 mais à frente neste capítulo.

8.3 Tag ou função?

Ao tratarmos da tag `<g:createLink>` 8.2 você deve ter observado que a usamos como se fosse uma função qualquer. Essa é uma das grandes vantagens por trás do modo como a tecnologia GSP manipula tags customizadas: você pode usá-las como se fossem funções. Não apenas em arquivos GSP, mas também em controladores e dentro das suas próprias bibliotecas de tag também. Você não precisa mais ficar aninhando tags tal como faria em JSP. Lembra como era?

```
<%-- Primeiro tínhamos de declarar a taglib... --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%-- Quanto sofrimento! E agora aninhá-la! --%>
<a href="<c:url value='/item/list'/>">
  Meu link
 </a>
```

Toda tag customizada em Grails tem como retorno um objeto do tipo `org.codehaus.groovy.grails.web.util.StreamCharBuffer` que, ao ser impresso, acaba nos retornando uma string. Imagine que em um de nossos controladores precisemos saber o endereço de uma URL gerada pela tag `<g:createLink>`. Como faríamos?

```
def actionQualquer() {
    // Armazeno a URL na variável endereço
    def endereco = createLink(controller:'cotacao')
}
```

Fig. 8.8: Atenção redobrada!

O modo como declarei a chamada à tag no controlador **não é recomendado** pois pode gerar algum conflito de nome em seu interior (imagine uma outra action com o nome `createLink` na mesma classe). O ideal é sempre preceder o nome da tag pelo seu namespace. Sendo assim, vamos reescrever este código:

```
def actionQualquer() {
    // Armazeno a URL na variável endereço
    // Agora usando o namespace correto!
    def endereco = g.createLink(controller:'cotacao')
}
```

Nem toda tag customizada em Grails possui o mesmo namespace `g:`. Adiante neste capítulo quando lhe ensinar como criar suas próprias *tag libraries* mostrarei como definir o seu próprio, o que é sempre uma boa prática. Apenas como exemplo, chamaremos uma tag customizada (e imaginária neste momento) do *ConCot* (`<concot:detalhesItem>`):

```
def detalhesItem = concot.detalhesItem(item:item)
```

8.4 Lidando com formulários

Claro, Grails também nos fornece uma série de tags que facilitam a vida daqueles que precisam lidar com formulários, ou seja, todos nós. As tags fundamentais nós já vimos ao tratarmos controladores 7, mas é importante que nos lembremos delas agora.

A principal e de longe mais usada é `<g:form>`, para declararmos formulários. Em sua essência, sua principal função é garantir que o atributo `action` da tag `<form>` seja preenchido corretamente. Os atributos mais importantes usados nesta tag portanto são:

- `controller` o nome do controlador que receberá a requisição (opcional caso estejamos lidando com uma página que esteja em um diretório cujo nome corresponda ao do controlador)
- `action` o nome da action para a qual faremos a submissão (obrigatório)
- `name` opcional: preenche o valor do atributo `id` da tag `<form>`.
- `useToken` usado para lidarmos com submissões duplicadas do formulário, conforme vimos no capítulo anterior 7.8.

A seguir podemos ver um exemplo simples da sua aplicação:

```
<g:form action="save" controller="cotacao">
<%-- Conteúdo do formulário entra aqui --%>
</g:form>
```

Outra variante importante é `<g:uploadForm>`, que possui comportamento praticamente idêntico ao de `<g:form>`, a diferença é que irá renderizar um formulário para submissão de arquivos, sendo que o preenchimento do atributo `enctype` é com o valor `"multipart/form-data"`. Um exemplo simples a seguir:

```
<g:uploadForm action="save" controller="item">
<input type="file" name="file"/>
</g:uploadForm>
```

Controles de entrada

Provavelmente os principais controles de entrada usados em qualquer formulário são os textuais. Para tal, Grails nos fornece duas tags: `<g:textField>` e `<g:textArea>`. Ambos possuem dois atributos essenciais: `name` e `value`, representando respectivamente o nome daquele campo e o valor neste armazenado. Seu uso é bastante direto:

```
<g:form action="save">
  <label for="nome">Nome:</label>
  <g:textField name="nome" value="${item?.nome}"/><br/>
  <label for="descricao">Descrição:</label>
  <g:textarea name="descricao" value="${item?.descricao}"/>
</g:form>
```

Estes campos irão renderizar, respectivamente, as tags `<input type="text" name="nome" id="nome"/>` e `<textarea name="descricao" id="descricao"></textarea>`.

Para renderizarmos um campo para preenchimento de senhas (`<input type="password"/>`) usamos a tag `<g:passwordField>`, que também possui os atributos `name` e `value`. Podemos ver um exemplo de sua aplicação no formulário de autenticação do *ConCot*:

```
<g:form action="autenticar" controller="autenticacao">
    <label for="login">Login:</label><br/>
    <g:textField name="login"/><br/>
    <label for="senha">Senha:</label><br/>
    <g:passwordField name="senha"/><br/>
    <input type="submit" value="Entrar"/>
</g:form>
```

Finalmente, temos também a renderização de caixas de seleção (tag `<select>`). Para isso, Grails nos fornece a tag `<g:select>`, que nos permite tratar esta tarefa (que pode ser bastante trabalhosa) de uma forma bastante simples.

A seguir, podemos ver um exemplo bastante simples, usado no formulário de cadastro de itens do *ConCot*, no qual o usuário seleciona qual a categoria a que um dado item a ser cadastrado pertence.

```
<g:form action="save">
  <label for="categoria.id">Categoria:</label>
  <g:select name="categoria.id"
            from="${categoriaList}"
            optionKey="id"
            optionValue="nome"
            value="${item?.categoria?.id}"/>
```

```
    <%-- Restante do formulário omitido --%>
</g:form>
```

Entender o significado dos parâmetros desta tag facilitará a compreensão do seu funcionamento.

- `name` Obrigatório, define o nome do item.
- `from` A lista sobre a qual a tag irá iterar gerando as tags `<option>` internas à `<select>`.
- `optionKey` Qual o valor que será armazenado no atributo `value` de cada elemento `<option>`.
- `optionValue` Qual o valor textual que deverá ser exposto em cada uma das opções da caixa de seleção.
- `value` Quando preenchido, diz qual é o item selecionado pela tag `<select>` após esta ter sido renderizada.

Também temos a tag `<g:checkBox>`, que é usada para renderizar caixas de seleção. Este tipo de campo de entrada normalmente é vinculado a atributos booleanos de nossas classes de domínio ou *command object*. A seguir podemos ver um exemplo aplicado em um cadastro de usuários:

```
<g:form action="save" controller="usuario">
<%-- Restante do formulário omitido --%>
<label for="ativo">Ativo:</label>
<g:checkBox name="ativo" checked="${usuario?.ativo}"/>
</g:form>
```

O atributo `checked` recebe como parâmetro uma expressão booleana que indica se o item encontra-se marcado como positivo ou negativo.

É importante mencionar que você não precisa usar as tags de componentes de entrada fornecidas pelo Grails, no entanto é interessante para sua aplicação dado que muitas vezes (como no caso de `<g:select>`) estas acabam por facilitar bastante a vida do programador. O formulário de autenticação a seguir, por exemplo, é perfeitamente válido:

```
<g:form action="autenticar" controller="autenticacao">
  <label for="login">Login:</label><br/>
  <input type="text" name="login"/><br/>
  <label for="senha">Senha:</label><br/>
  <input type="password" name="senha"/><br/>
  <input type="submit" value="Entrar"/>
</g:form>
```

8.5 Criando suas próprias tags customizadas

Uma biblioteca de tags nada mais é que uma classe Groovy que segue algumas convenções:

- Encontra-se armazenada no diretório `grails-app/taglib`.
- Seu nome possui o sufixo `TagLib`.
- Cada tag em seu interior é representada por uma closure.

Há um comando do CLI do Grails que gerará este arquivo de forma automática para você (lembre-se, é apenas uma comodidade, seguindo essas convenções você terá o mesmo resultado): trata-se de `create-tag-lib`. Vamos usá-lo para gerar a biblioteca de tags `Concot`:

```
create-tag-lib Concot
```

Finalizada sua execução, serão gerados dois arquivos: `grails-app/taglib/concot/ConcotTagLib.groovy` (o pacote padrão é o nome da sua aplicação) e `test/unit/concot/ConcotTagLibSpec.groovy`, que é o esqueleto para que você escreva seus testes unitários para sua *tag library*.

O arquivo gerado é bastante simples:

```
package concot

class ConcotTagLib {
    static defaultEncodeAs = [taglib:'html']

}
```

Segurança

Fig. 8.9: Atenção redobrada!

O atributo estático `defaultEncodeAs` define como o conteúdo gerado pela nossa biblioteca deverá ser renderizado pela camada de visualização. O valor padrão, como pode ser visto, é `'html'`. Isso permite que uma série de falhas de segurança relacionadas a XSS [25] seja evitada.

O valor padrão para cada tag definida em nossa biblioteca é `'html'`. A razão é simples: segurança. Imagine que um usuário mal intencionado digite como descrição de um item no *ConCot* um texto similar ao seguinte:

```
<script type="text/javascript">
    while(true)
    {alert('Travei sua tela')}
</script>
```

Se não houver nenhum tratamento durante a renderização do conteúdo, este texto será incluído em todas as páginas que expuserem o atributo `descricao` daquele item. Resultado? Sua página irá travar em um loop infinito. Como resolvemos isso? Com o atributo `defaultEncodeAs`, que irá "escapar" o texto a ser renderizado, de tal modo que teremos algo similar ao exemplo a seguir durante a renderização:

```
&lt;script type="text/javascript"%gt
    while(true)
    {alert('Travei sua tela')}
&lt;/script%gt;
```

Você está protegido agora. Por padrão, mantenha esta configuração, a não ser que você saiba exatamente o que está fazendo. Caso queira expor o texto puro (e evitar assim o escape de caracteres), basta passar o valor `raw` para este atributo.

Nossa primeira tag

Antes de começar, é sempre uma boa prática definirmos qual o namespace adotado pela nossa *tag library*. Essa é uma excelente prática pois evita conflitos de nomes de tags. Fazemos isso definindo o atributo estático `namespace`, tal como na primeira evolução do nosso arquivo `ConcotTagLib.groovy`.

```
package concot

class ConcotTagLib {
    // Nosso namespace
    static namespace = 'concot'

    static defaultEncodeAs = [taglib:'html']

}
```

Nossa primeira tag irá gerar uma tag `<a>` com um link que apontará para a action `imagem` do controlador `ImagemController` que implementamos em nosso capítulo sobre controladores 7.10. Esta nossa tag se chamará `imagem`, e vai gerar como saída código HTML que sabemos ser seguro. Sendo assim, nosso primeiro passo é incrementar o atributo `defaultEncodeAs` para que fique similar à listagem a seguir:

```
class ConcotTagLib {

    static namespace = 'concot'

    static defaultEncodeAs = [taglib:'html', imagem:'raw']

}
```

Em seguida, entra em ação a primeira versão da nossa tag:

```
/*
  Tag que gera uma imagem para um
  @attr item Objeto do tipo Item cuja imagem será renderizada caso exista
*/
def imagem = {attrs, body ->
```

```
        if (attrs.item?.imagem) {
            def link = g.createLink(controller:'item', action:'imagem', id:
            out << "<img src=\"${link}\"/>"
        }
}
```

Tudo o que fizemos foi declarar uma closure em nossa classe chamada `imagem`, que espera dois parâmetros: `attrs` e `body`. Neste momento nos interessa apenas o primeiro, que representa o que passaremos à nossa tag. Como pode ser observado na listagem, trata-se de um map convencional que pode armazenar qualquer tipo de informação.

A variável `out` é disponibilizada para todas as bibliotecas de tag e representa o stream de saída dela. No caso, o operador `<<` foi sobrescrito: o que vier do lado direito será "transferido" para o stream de saída, que irá, em seguida, transmitir o conteúdo para o cliente da nossa aplicação. Observe que tiramos proveito da tag `<g:createLink>` em seu interior sob a forma de função, expondo mais uma vez a vantagem desta possibilidade.

É muito recomendável que você inclua comentários em formato JavaDoc em sua action, pois além do ganho óbvio, eles irão ajudar o suporte a *autocomplete* de IDEs como JetBrains IDEA e GGTS (Groovy e Grails Tool Suite). Basta escrever comentários como o que fiz, listando cada atributo esperado pela tag precedido por `@attr`.

O próximo passo é a aplicação da nossa tag em nossos arquivos GSP. O primeiro local onde a inseri foi na página `show.gsp` usada pelo controlador `item`.

```
<%--
    itemInstance é uma das variáveis
    definidas no modelo da action show de
     ItemController
--%>
<concot:imagem item="${itemInstance}"/>
```

Repare que, como mencionado, foi um procedimento extremamente simples usar a biblioteca: não precisamos incluir nenhuma instrução de importação como no JSP nem nada parecido. Basta usar a tag! O resultado durante a renderização ficou... "ok", para este primeiro momento.

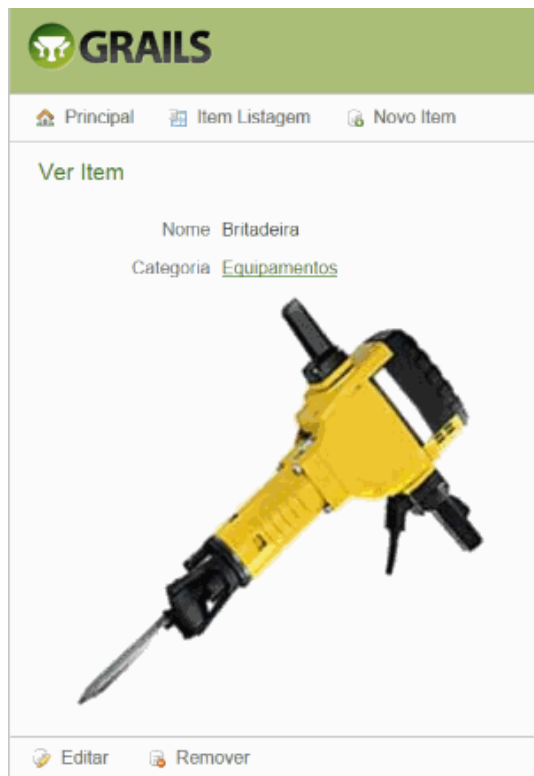



Fig. 8.10: Nossa primeira tag

Infelizmente nossa tag não possui atributos para definir o tamanho da imagem a ser renderizada. Fica como exercício para você implementar esta funcionalidade. Dica: basta esperar mais um atributo. :)

E aquele parâmetro body?

Nada mais é do que o texto que nossas tags customizadas envolvem. Que tal aprimorarmos um pouco mais nossa tag `<concot:imagem>` para que tire proveito deste atributo? Que tal expor o texto envolvido pela tag caso a instância de `Item` passada como parâmetro não possua o campo `imagem` preenchido? Fácil de resolver!

```
def imagem = {attrs, body ->
       if (attrs.item?.imagem) {
           def link = g.createLink(controller:'item', action:'imagem', id:
           out << "<img src=\"${link}\"/>"
       } else {
        // simplesmente passamos a execução de corpo()
        // para ser renderizada!
        out << body()
    }
}
```

Repare que `body` na realidade é uma closure que pode ser executada em nossa tag. Como aplicaríamos esta tag em nosso sistema agora?

```
<concot:imagem item="${itemInstance}">
  <p>Item sem imagem</p>
</concot>
```

O texto `<p>Item sem imagem</p>` será renderizado no lugar da tag `<a>`. Bem simples, concorda? Como segundo exercício, que tal tentar implementar a tag `<g:if>`?

```
def ifConcot = {attrs, body ->
    if (attrs.test) {
        out << body()
    }
}
```

8.6 Templates

Templates são outra maneira de se componentizar elementos usados pela camada de visualização. Pense neles como pequenos trechos de GSP que você pode reaproveitar em mais de um ponto do seu projeto. Uma solução bem mais interessante que o nocivo, perigoso e velho "copiar e colar".

Criar um template é simples: basta que você salve seu arquivo GSP com o prefixo "_" em seu nome em alguma pasta dentro de `grails-app/views` de sua preferência. Para começarmos, vamos criar um template que nos possibilite padronizar o modo como itens são renderizados no *ConCot*.

Para tal, vamos criar um arquivo chamado `_item.gsp` no diretório `grails-app/views/item`:

```
<div class="item">
    <div class="categoria">${item?.categoria?.nome}</div>
    <div class="nome">
        <g:link action="show" controller="item" id="${item?.id}">
        ${item?.nome}
        </g:link>
        <concot:imagem item="${item}">
          <p>Sem imagem :(</p>
        </concot:imagem>
    </div>
</div>
```

Observe que nosso pequeno template espera seu próprio model, exatamente como em qualquer arquivo GSP. Há duas opções para se renderizar este conteúdo.

A primeira é através da tag `<g:render>`, que recebe dois parâmetros: o nome do template e o modelo a ser renderizado, como no exemplo a seguir:

```
<g:render template="/item/item"
          model="[item:itemInstance]"/>
```

Observe que não precisamos digitar o conteúdo do atributo `model` como se fosse uma expressão Groovy. Basta que exista uma variável chamada `itemInstance` no contexto em que nosso template está sendo renderizado. Outro ponto de atenção: você não precisa colocar o caractere "_" ao declarar o nome do template.

Fig. 8.11: Atenção redobrada!

Claro, dado que o template nada mais é do que outro arquivo GSP, você também pode usar a tag `<g:render>` no interior dele quando precisar criar

renderizações recursivas ou que usem outros templates em sua composição. **Cuidado com isto**. Templates são uma solução interessante de componentização, mas seu uso excessivo pode levar a código GSP **extremamente difícil** de ser mantido e entendido. Apenas se imagine decompondo mentalmente um processo de renderização envolvendo templates embarcados em outros templates.

A função `render`, que pode ser usada tanto em controladores quanto tags, também pode renderizar templates. Basta passar como parâmetro o template a ser renderizado e qual o modelo.

```
def renderItem(Item item) {
    render(template:'/item/item', model:[item:item])
}
```

8.7 Padronizando layouts com Sitemesh

Grails adotou desde sua primeira versão o Sitemesh como seu motor de layouts. Esse tipo de ferramenta é a responsável por nos ajudar a manter a identidade visual da nossa aplicação com o mínimo esforço possível.

Se você navegar pelas páginas do *ConCot* vai perceber que existe uma identidade visual composta por três elementos esquematizados na imagem a seguir:

Fig. 8.12: Entendendo o SiteMesh

A única parte que varia é o corpo, enquanto cabeçalho e rodapé se mantêm os mesmos sempre. Para obter este resultado podemos pensar em algumas soluções. A primeira delas poderia ser ter dois templates, representando o cabeçalho e o rodapé padrão do sistema, que simplesmente renderizaríamos em todas as páginas do sistema usando `<g:render>`. É uma solução ruim, pois é propensa a erros (você pode esquecer de incluir os templates) e trabalhosa (você precisará repetir o mesmo processo em todas as páginas do seu projeto).

Apenas por curiosidade, abra qualquer página GSP do *ConCot* neste momento e busque pelo trecho HTML no qual o logotipo do Grails é incluído no cabeçalho das páginas. Você não o encontrará, mas talvez você perceba um trecho interessante no corpo da tag `<head>`:

```
<meta name="layout" content="main"/>
```

Tudo ficará mais claro até o final desta seção. ;)

É o Sitemesh por trás dos panos que resolve a questão da uniformidade visual em projetos Grails. Tudo começa com a criação de um arquivo de layout, no qual iremos definir os padrões visuais que se repetirão em todas as páginas do *ConCot*. Por padrão, todos os arquivos de layout são incluídos no diretório `grails-app/views/layouts`. Quando criamos nosso projeto, foi criado também o default da aplicação que é o arquivo `main.gsp` (lembra do corpo da tag `<head>`?), cuja versão simplificada para fins didáticos pode ser vista a seguir:

```
<html>
  <head>
    <title><g:layoutTitle default="Grails"/></title>
    <g:layoutHead/>
  </head>
  <body>
    <div id="grailsLogo" role="banner">
        <%-- O logotipo do Grails está aqui.
             Foi removido da listagem apenas
             para simplificar sua leitura --%>
    </div>
    <g:layoutBody/>
```

```
  </body>
</html>
```

Encontramos o local onde o logotipo do Grails se encontra: o arquivo main.gsp. Como você deve ter observado, é como se este arquivo fosse fundido com todas as nossas outras páginas. Na realidade, não é "como se": fundir páginas é exatamente o que o Sitemesh faz.

A instrução `<meta name="layout" content="main"/>` diz para o Sitemesh aplicar o layout definido no arquivo main.gsp na página corrente. Vamos analisar o conteúdo deste arquivo de layout, para que as coisas fiquem mais claras.

<g:layoutTitle>

No arquivo main.gsp podemos observar a presença de algumas novas tags. A primeira é `<g:layoutTitle>`, com o texto Grails passado como valor para o parâmetro default. Seu funcionamento é simples: se na página sobre a qual será aplicada o layout não houver a tag `<title>` preenchida, o valor selecionado será Grails, caso contrário, o valor definido na página alvo.

<g:layoutHead>

Já a tag `<g:layoutHead>` irá incluir no arquivo de layout todo o conteúdo da tag `<head>` presente na página alvo, enquanto `<g:layoutBody>` irá fazer exatamente o mesmo trabalho: a diferença está agora no fato de que será inserido no arquivo de layout o conteúdo da tag `<body>` da página alvo. O resultado? As duas páginas fundidas são enviadas como resposta ao usuário final.

Observe que você não precisou nas páginas alvo escrever alguma notação estranha: apenas o HTML necessário, mantendo a sintaxe do HTML intacto e facilitando o trabalho da sua equipe, que não precisa aprender nenhuma outra linguagem ou ficar alterando arquivos de configuração, tal como ocorre em outras soluções como o Apache Tiles [11] (outro motor de layouts bastante popular, especialmente entre desenvolvedores Spring e Struts).

Ativando layouts

Há mais de uma maneira de se ativar layouts do Sitemesh no Grails. A primeira você acabou de ver: basta incluir a tag <meta name="layout" content="nome do layout"/> no corpo da tag <head> da sua página. Lembre-se que não é preciso incluir a terminação do arquivo no interior do atributo content.

É importante mencionar que você pode criar quantas pastas quiser no interior de grails-app/views/layouts. Sendo assim, caso exista um arquivo chamado grails-app/views/layouts/principal/main.gsp, você o referenciaria como

```
<meta name="layout" content="/principal/main"/>
```

Você também pode definir o layout padrão para todas as actions presentes em um controlador. Para isso, basta incluir no código-fonte deste controlador o atributo estático layout recebendo como valor o nome do layout a ser aplicado:

```
class ItemController {
    static layout = "main"
}
```

Fazendo isso, não é necessário incluir a tag <meta name="layout"> em suas páginas GSP.

Finalmente, a terceira opção é basear-se em convenções do Grails. Se houver no interior da pasta de layouts um arquivo cujo nome seja igual ao do controlador, ele será aplicado às páginas renderizadas por este.

Sendo assim, se por exemplo, existir o arquivo grails-app/views/layouts/item.gsp, e não for definido o atributo estático layout ou a tag <meta> em seus arquivos GSP, este será o layout padrão. Ainda melhor, você pode também definir o layout para uma action específica, basta que o arquivo siga a seguinte convenção em seu local de armazenamento e nome:

```
/views/layouts/[nome do controlador]/nomeDaAction.gsp
```

A mesma regra se aplica: o arquivo `grails-app/views/layouts/item/create.gsp` poderia ser o layout padrão da página renderizada pela action `create` do controlador `item`. Novamente, apenas se não for criado o atributo estático `layout` ou for incluída a tag `<meta>` no arquivo GSP.

8.8 Recursos estáticos

A manipulação de recursos estáticos em uma aplicação web é uma das tarefas mais menosprezadas pelos desenvolvedores e também uma das que mais pode penalizar o desempenho e consumo de banda do projeto. Será que a melhor maneira de referenciar recursos estáticos como arquivos JavaScript, CSS e imagens é realmente apenas usar a tag `<g:resources>` que vimos neste capítulo? Existe alternativa melhor? Yeap!

Há diversas oportunidades de otimização que podem ser levadas em consideração ao transmitirmos recursos estáticos para nossos clientes:

- Podemos minimizar nossos arquivos CSS e JavaScript antes do envio, reduzindo o consumo da banda.
- Ainda melhor: para evitar as diversas requisições que podem chegar ao nosso servidor, podemos simplesmente combinar vários destes arquivos em um só e enviar todos minimizados de uma única vez.
- Muito conteúdo só precisa ser enviado uma vez. Pense no logotipo do seu site. Será que o usuário realmente precisa baixá-lo todas as vezes? É possível enviar cabeçalhos HTTP para o cliente que informem a data de expiração daquele recurso, evitando que novas requisições sejam feitas ao servidor para baixá-lo.

Todos estes problemas (e mais alguns) são resolvidos pelo plugin `Asset Pipeline` que já vem por padrão com o Grails desde a versão 2.4.0 e que é o assunto desta seção. O primeiro ponto de atenção são três diretórios que são criados com todo novo projeto Grails que representam os locais onde o plugin irá buscar arquivos de imagens, JavaScript e CSS:

- grails-app/assets/images
- grails-app/assets/javascripts
- grails-app/assets/stylesheets

Idealmente, todo recurso estático do seu projeto deve ser armazenado nestas pastas visando com isto tirar proveito das otimizações trazidas pelo Asset Pipeline. *E como eu faço para começar a usar o plugin?* Simples, através de algumas tags customizadas voltadas para esses três tipos de recursos.

Para incluir seus recursos, basta usar as tags `<asset:javascript/>`, `<asset:stylesheet>` e `<asset:image>`. Todas possuem o parâmetro `src`, cujo valor deverá ser o nome do arquivo a ser transmitido ao cliente e cujo caminho é relativo ao diretório raiz no qual é armazenado. Seguem alguns exemplos:

```
<%-- Incluindo o arquivo CSS
     /views/assets/stylesheets/estilo.css
    <link rel="stylesheet" href="assets/stylesheets/estilo.css"/>
--%>
<asset:stylesheet src="estilo.css"/>
<%-- Incluindo o arquivo JavaScript
    /views/assets/javascripts/scripts.js
    <script src="assets/javascripts/script.js">
    </script>
--%>
<asset:javascript src="scripts.js"/>
```

Nos dois exemplos, os seguintes processamentos serão executados sobre os arquivos transmitidos:

- Será enviado o cabeçalho HTTP `Expires`, informando ao browser que este poderá cachear estes recursos, evitando subsequentes consultas ao seu servidor.
- No caso de conteúdo textual (CSS, JavaScript), este será minimizado, reduzindo o tamanho dos arquivos baixados pelo navegador.

- No caso de bundles (veremos mais a respeito a seguinte), estes serão montados e fornecidos ao servidor.

A única diferença está na tag `<asset:image>`, que pode receber todos os atributos que normalmente passaríamos para a tag HTML `<img>` como, por exemplo, `id`, `style`, `class`, `width`, `height` ou qualquer outro que você deseje, tal como nos exemplos a seguir:

```
<%-- Incluindo a imagem
    /views/assets/images/logo.png --%>
<asset:image src="logo.png"/>
<%-- Adicionando mais alguns atributos opcionais --%>
<asset:image src="logo.png"
             class="logo"
             width="450px"
             style="border-style:none"
             id="idLogo"/>
```

Criando bundles

Muitas vezes nossos recursos estáticos possuem dependências entre si. Isto é muito comum quando lidamos com JavaScript, por exemplo. Imagine que tenhamos um arquivo chamado `scripts.js` que use a biblioteca jQuery, armazenada no arquivo `jquery.js`. Sem o uso do Asset Pipeline, para que nosso script funcionasse corretamente, precisaríamos escrever código similar ao abaixo:

```
<script type="text/javascript"
        src="${resource(dir:'js', file:'jquery.js')}">
</script>
<script type="text/javascript"
        src="${resource(dir:'js', file:'script.js')}">
</script>
```

Trata-se de uma maneira bastante ineficiente de se referenciar nossos recursos. Primeiro, porque foram necessárias duas requisições ao nosso servidor (uma para cada arquivo), e segundo, porque não há qualquer informação sobre cacheamento sendo fornecida ao navegador. Uma solução interessante

seria juntar o conteúdo dos dois arquivos em um só, mas esta é uma tarefa propensa a erros, pois você poderia simplesmente esquecer de fazê-lo caso não automatizasse o processo.

Mas o que é um bundle?

Nada mais do que um conjunto de recursos estáticos interdependentes que são combinados em um único arquivo.

Uma alternativa muito mais interessante é armazenar os dois arquivos no diretório apropriado lido pelo Asset Pipeline (`grails-app/views/assets/javascripts`) e em seguida declarar as dependências em um manifesto como o seguinte, que chamaremos de `bundle.js`.

```
/**
//= require jquery.js
//= require script.js
*/
```

A instrução `//= require` inclui em nosso bundle o conteúdo do arquivo referenciado (observe que o caminho ao arquivo é relativo). Em seguida, basta referenciar nosso bundle em nosso arquivo GSP tal como no exemplo a seguir:

```
<asset:javascript src="bundle.js"/>
```

Em tempo de execução, os dois recursos referenciados serão incluídos e minimizados em um bundle, que será transmitido para o cliente incluindo o cabeçalho HTTP `Expires`, permitindo que nosso navegador cacheie aquele recurso, evitando futuras requisições.

É interessante observar que todo este processamento só ocorre no ambiente de produção, o que proporciona ao desenvolvedor continuar alterando seus recursos estáticos durante desenvolvimento, e facilita enormemente o processo, pois o bundle será gerado a cada requisição.

Aliás, no momento de geração do arquivo `WAR` todos os bundles são pré-compilados, evitando que em produção este processamento ocorra novamente. Isso aumenta o desempenho da aplicação e reduz o tempo de inicialização no container web.

A instrução `//= require` é o que chamamos de *diretiva de importação*. Um conjunto de diretivas formam o que, no jargão do Asset Plugin é chamado de manifesto (*manifest*). O prefixo usado na escrita da diretiva varia de acordo com o formato do recurso estático:

- **CSS** prefixo `*=`
- **JavaScript** prefixo `//=`

Há outras três diretivas além de `require`:

- `require_self` inclui o conteúdo do arquivo no qual a diretiva é incluída no bundle.
- `require_tree` incluirá todo o conteúdo da árvore na qual o arquivo de bundle se encontra.
- `require_full_tree` incluirá todo o conteúdo da árvore na qual o arquivo de bundle se encontra e também de todos os plugins que possuam arquivos no mesmo diretório em sua estrutura. Veremos mais sobre plugins em um capítulo dedicado ao assunto neste livro 11.

Voltando ao nosso bundle inicial, poderíamos reescrevê-lo usando apenas os arquivos `script.js` e `jquery.js`, como no exemplo a seguir:

```
/**
//= require jquery.js
//= require_self
*/

function processeItens() {
    /*esta função poderia estar usando
      as funções do jQuery sem problema agora. */
}
```

Plugin Resources

Fig. 8.13: Compatibilidade

Da versão 2.0.x até a 2.3.x, o plugin padrão usado pelo Grails para gerenciar a transmissão de recursos estáticos era o Resources [19], que a partir da versão 2.4.0 foi substituído pelo Asset Pipeline. Este será inclusive o padrão adotado na versão 3.0 do framework.

Esta substituição trouxe uma série de vantagens ao Grails como, por exemplo, um número menor de dependências externas, manipulação de bundles (veremos mais à frente) mais simples e melhorias de desempenho.

É importante mencionar que o Grails a partir da versão 2.2.0 executado sobre a JVM 7+ já é compatível com o Asset Pipeline. Se você possui um projeto baseado nesta versão (ou posterior) do Grails a substituição do antigo plugin Resources pelo Asset Pipeline é um excelente negócio pois irá reduzir seu trabalho durante o upgrade para a versão 3.0 do framework.

8.9 Ajax

Grails oferece suporte nativo a Ajax através de um pequeno conjunto de tags, que tornam a aplicação desta prática uma tarefa trivial. Para ilustrar sua aplicação, vamos construir um formulário de busca de itens no *ConCot*, substituindo sua página inicial, cujo resultado final será similar ao exposto da imagem:

Fig. 8.14: Aplicando AJAX

O primeiro passo é incluir a tag `<g:javascript>` no interior de `<head>`:

```
<head>
  <g:javascript library="jquery"/>
</head>
```

Isso irá ativar o suporte a Ajax usando a biblioteca jQuery, que já vem por padrão instalada com o Grails. É possível usar outras bibliotecas além do jQuery, como por exemplo Dojo, no entanto neste livro vamos falar apenas a respeito da biblioteca padrão usada pelo Grails 2.4.

Atenção para o Asset Pipeline

Fig. 8.15: Compatibilidade

Caso você tenha migrado sua aplicação para a versão 2.4 do Grails e removido o plugin Resources, você deve evitar usar a instrução `<g:javascript>`. Em seu lugar, use `<asset:javascript>` tal como no exemplo a seguir:

```
<head>
  <asset:javascript src="jquery.js"/>
<head>
```

Esse código força o carregamento da biblioteca usando o plugin Asset Pipeline.

Nosso próximo passo consistirá na escrita do formulário Ajax usando a tag `<g:remoteForm>`:

```
<g:formRemote name="formBusca" update="resultado"
              url="[controller:'item', action:'busca']">
  Buscar item:
  <g:textField name="nome"/>
  <input type="submit" value="Buscar"/>
</g:formRemote>

<div id="resultado"></div>
```

No atributo `update` definimos o identificador da tag HTML que receberá o resultado da nossa consulta. O atributo `url` define qual a action que deverá ser executada. Note que passamos um mapa como valor e que o conteúdo do

formulário não possui nada de diferente se comparado com um formulário convencional.

Nossa action `busca` é bastante simples: apenas renderiza um template como resultado. Veja a seguir:

```
def busca() {
  def itens = Item.findAllByNomeLike("%${params['nome']}%")
  render(template:'resultado', model:[itens:itens])
}
```

O template `grails-app/views/item/_resultado.gsp` é igualmente simples.

```
<table>
 <thead>
     <th>Nome</th>
     <th>Imagem</th>
 </thead>
 <tbody>
  <g:each in="${itens}" var="item" status="i">
   <tr class="${i % 2 == 0 ? 'even' : 'odd'}">
      <td>
     <g:link action="show" id="${item.id}">${item.nome}</g:link>
    </td>
      <td>
     <concot:imagem item="${item}"/>
    </td>
   </tr>
  </g:each>
 </tbody>
</table>
```

Voilà: nosso formulário Ajax está pronto para ser usado e funcionará perfeitamente. Outra tag interessante que surte efeito similar é `<g:remoteLink>`. Ela funciona de uma forma muito parecida com a tag `<g:link>`. A diferença está na presença do atributo `udpate` que, assim como no caso de `<g:formRemote>`, recebe como valor o identificador da tag HTML na qual deverá ser renderizada a resposta do processamento da nossa action.

Tags Ajax consideradas obsoletas

Fig. 8.16: Compatibilidade

Na documentação da versão 2.4.0 [10] é dito que as tags `<g:formRemote>`, `<g:remoteField>`, `<g:remoteFunction>` e `<g:remoteLink>` devem ser consideradas obsoletas (*deprecated*) e que seriam removidas de futuras versões do Grails. Caso esteja pensando em migrar seu projeto para Grails 3.0, este é um ponto que deve ser levado em consideração.

Isso não quer dizer que as tags deixarão de existir. Preocupado com isso, enviei um e-mail para a lista de discussão usada pelos desenvolvedores do framework [15] e fui tranquilizado pelo próprio Graeme Rocher, principal responsável pela evolução e manutenção do Grails, que nos disse que essas tags na realidade serão movidas para um plugin. Sendo assim, quando a versão 3.0 do Grails sair, bastará usar este plugin, que ainda não foi publicado.

8.10 Customizando o scaffolding

Finalmente, uma última dica: você pode customizar 100% do scaffolding do Grails, sabia? Para tal, basta executar o comando `install-templates` na interface de linha de comando. Isso irá salvar na pasta `src/templates`.

O conteúdo armazenado em `src/templates` na realidade são os mesmos usados pelos comandos `create-domain-class`, `create-controller`, `create-tag-lib`, `generate-views`, `generate-all` e basicamente todos os artefatos que conseguimos

construir através da linha de comando do framework.

O mais interessante é que podemos customizar o modo como o scaffolding é gerado também. Basta modificar o conteúdo da pasta `src/templates/scaffolding`. Com isso é possível, dentre outras coisas, modificá-los de tal modo que passem a usar algum framework CSS como, por exemplo, Twitter Bootstrap ou Zurb Foundation.

Bug na versão 2.4.4 do Grails

Fig. 8.17: Atenção redobrada!

Infelizmente, há um bug na versão 2.4.4 do Grails que não permite que os templates usados na geração dos arquivos GSP (os arquivos mais customizados pelos desenvolvedores!) não sejam incluídos no diretório `src/templates`. Felizmente este é um problema fácil de ser resolvido.

No diretório de instalação do Grails, busque pelo arquivo `plugins/scaffolding-2.1.2.zip`. Descompacte seu conteúdo em uma pasta qualquer de sua preferência, você encontrará uma pasta chamada `scaffolding` no diretório `src/templates`. Basta copiá-la para a pasta `src/templates` do seu projeto e o problema está resolvido: pode customizar seu scaffolding agora. ;)

Capítulo 9

Camada de negócio: serviços

O código mais importante em todo sistema é aquele no qual escrevemos a lógica de negócio do cliente. Arquiteturalmente, este componente deve se posicionar da forma mais isolada possível, ou seja, não deve possuir dependência alguma com as camadas de controle e visualização: idealmente, teria apenas com a camada de persistência.

Grails nos incentiva a implementar a lógica de negócio em serviços (*services*). Neste capítulo vamos entender sua natureza e com isso conhecer a essência do próprio Grails que é o Spring Framework [31].

9.1 Serviços

Serviços (*services*) são a solução fornecida pela equipe de desenvolvimento do Grails para que o programador possa implementar da forma mais isolada possível as regras de negócio do seu projeto. Mas serviços não são apenas

"código isolado": eles também tiram proveito do Spring Framework, que nos fornece poderosos recursos como inversão de controle, injeção de dependências, transacionalidade e AOP.

Mas por que não devemos implementar regras de negócio em controladores?

- Você perde o isolamento da regra de negócio, que passa a ficar acoplada à camada de controle e visualização. Normalmente, isso é sintoma de um acoplamento ainda maior. Exemplo: você altera sua página GSP e precisa mudar o controlador, aí de repente percebe que precisa alterar aquela regra de negócio também. A situação lhe soa familiar?
- É perdida a separação de responsabilidades. Seu controlador passa a ser responsável tanto por gerenciar as requisições recebidas pela sua aplicação quanto a regra de negócios do projeto (veja o ponto acima).
- A testabilidade fica comprometida. É muito mais fácil testarmos um serviço isoladamente pois o número de dependências é menor.
- O reúso é comprometido: como acessar o código presente em um controlador a partir de outras classes que não sejam controladores?
- Dado que a separação de responsabilidades é perdida, normalmente a complexidade do código presente nos controladores aumenta significativamente.

Vamos escrever nosso primeiro serviço?

Nosso primeiro serviço

No capítulo 7 vimos um primeiro candidato a serviço. Lembra quando implementamos nosso comunicador com os fornecedores 7.7? Podemos implementá-lo agora como um serviço. Novamente, duas convenções devem ser seguidas.

- A classe deve estar no diretório `grails-app/services`
- Deve ter em seu nome o sufixo `Service`

Você também pode criá-la usando a interface de linha de comando. Basta executar o comando `create-service Comunicacao`:

```
create service Comunicacao
```

O comando irá criar dois arquivos: `grails-app/services/concot/ComunicacaoService.groovy` e `test/unit/concot/ComunicacaoServiceSpec.groovy`, onde escreveremos nossos testes. Neste primeiro momento, apenas o primeiro nos interessa:

```
package concot

import grails.transaction.Transactional

@Transactional
class ComunicacaoService {
    def serviceMethod() {

    }
}
```

É uma classe como outra qualquer: a diferença está na presença da anotação `@Transactional` sobre a qual falaremos com detalhes mais à frente. É inclusive incluído um método de exemplo que, naturalmente, iremos apagar (para que um método chamado *serviceMethod*?).

Nossa primeira versão deste mecanismo é similar à exposta a seguir. Não se preocupe agora com o que colocaremos no interior do método `enviarMensagem`. Primeiro vamos usar (ou melhor, injetar) o serviço onde se faz necessário.

```
@Transactional
class ComunicacaoService {
    def enviarMensagem(Fornecedor fornecedor,
                       String email,
                       String mensagem) {
        // Executa a tarefa de envio de e-mail
    }
}
```

Injeção de dependências: usando o serviço

Entra em cena o conceito de injeção de dependências. Se você nunca trabalhou com Spring, acredito que a melhor maneira de entender o termo é a prática. Primeiro vamos fazer duas pequenas mudanças na classe `FornecedorController`.

```
class FornecedorController {
  // restante da classe omitido

  def comunicacaoService

  def enviarMensagem(EnvioEmail envio) {
      envio.validate()
      if (envio.hasErrors()) {
          // tratamos o erro
      } else {
          flash.message = "Mensagem enviada com sucesso"
        // Usamos nosso serviço
          comunicacaoService.enviarMensagem(envio.fornecedor,
                                            envio.email,
                                            envio.mensagem)
          render(view:'comunicacao')
      }
  }
}
```

Ao declararmos um atributo em nosso controlador cujo nome coincida com o do serviço que acabamos de criar (`comunicacaoService`), em tempo de execução teremos iniciado o processo de injeção de dependências, ou seja, este atributo irá receber como valor uma instância do nosso serviço quando o controlador for instanciado e estará pronto para uso, que é exatamente o que faremos na action `enviarMensagem`.

E não é só em controladores que a injeção de dependências funciona: também podemos injetar serviços em classes de domínio, *tag libraries* e *command objects*, o que expõe bem o poder de componentização dos serviços.

Como esta instância do serviço é criada e injetada? Primeiro lhe direi *quem* cuida da instanciação e injeção: o Spring Framework. Chamamos Grails de

framework mas na realidade estamos lidando com um *meta-framework*, isto porque uma aplicação Grails nada mais é que uma fina camada Groovy construída para tirar máximo proveito do Spring.

Observe que você não precisou instanciar manualmente um objeto do tipo `Comunicacaoservice`, isto foi feito para você pelo container de injeção de dependências e inversão de controle do Spring, que cuidará não só da instanciação e injeção do objeto, mas também se preocupará em prover uma série de funcionalidades adicionais para nós, especialmente transacionalidade, que veremos adiante, de uma forma completamente transparente.

Container Spring?

O núcleo do Spring Framework é o que costumamos chamar de container de inversão de controle (*IoT Inversion of Control*) e injeção de dependências (*DI Dependency Injection*). Este componente é responsável por cuidar do ciclo de vida de um objeto, isto é: quando uma instância deve ser criada e onde deverá ser injetada em outros objetos.

O nome "inversão de controle" vem deste ato de delegarmos a responsabilidade pela instanciação dos objetos para o container. Você, programador, não precisa se preocupar em escrever código como `new ComunicacaoService()`: quem executa esta tarefa é o container, que também será o responsável pelo descarte destas instâncias para você (veremos mais sobre este "descarte" na próxima seção deste capítulo).

Estas instâncias gerenciadas pelo Spring são o que chamamos de *spring beans* ou simplesmente *beans*. Lembra quando disse agora há pouco que o Grails é na realidade um meta-framework e que você poderia injetar seus serviços em controladores, classes de domínio, bibliotecas de tag e *command objects*? Isso é possível porque todos estes elementos *também são beans do Spring*.

O termo "injeção de dependências" também merece uma rápida explicação. Quando na classe `FornecedorController` escrevemos o trecho:

```
class FornecedorController {
    def comunicacaoService
}
```

`comunicacaoService` é o que chamamos de uma *dependência da classe* *`FornecedorController`*. O ato de injetar dependências equivale ao processo executado pelo container do Spring de primeiro descobrir quais as dependências necessárias para a execução de um *bean* e em seguida injetá-las naquele objeto.

9.2 Escopos

Vimos que o container do Spring se encarrega de instanciar nossos serviços e também injetá-los para nós onde se fazem necessários, mas na realidade também resolve outro problema: quando o objeto deve ser descartado? Por quanto tempo uma instância deve existir? Depende do escopo.

Sabemos que o contexto padrão de um controlador Grails é *prototype* 7.12, ou seja, será criada uma nova instância que durará somente enquanto a requisição estiver ativa (a descrição que fiz naquele momento do escopo `prototype` é imprecisa, e você entenderá por que digo isto até o final desta seção).

O escopo atende duas finalidades: a mais óbvia é definir por quanto tempo uma instância deve existir. A menos evidente é evitar problemas de concorrência. Volto a repetir: uma aplicação web é uma aplicação intrinsecamente concorrente. O tempo inteiro mais de um usuário irá iniciar a execução de métodos em uma mesma classe, e cada instância, por sua vez, possui seu próprio estado interno. Como evitar que um usuário acesse ou modifique o estado de outro gerando erros difíceis de serem detectados e resolvidos? Usando o escopo correto.

A descrição dos escopos nos ajuda a entender melhor como o problema é resolvido.

- `session` Uma instância do objeto é criada e existirá apenas dentro do período de existência **e** estará acessível na sessão do usuário corrente. Sendo assim, havendo dois usuários no sistema, temos duas instâncias do mesmo objeto, uma para cada sessão, e não veremos problemas de concorrência.

- `prototype` Toda vez que o objeto for necessário, o Spring se encarregará de criar uma nova instância para o objeto que precisa daquele

recurso. *Quando o objeto é necessário?* No momento em que se instancia um novo bean e o Spring detecta suas dependências ou quando pedimos diretamente ao container do Spring uma nova instância do bean.

- `singleton` O mais perigoso e sedutor dos escopos. Uma única instância será criada e compartilhada por todas as requisições que nossa aplicação venha a receber. Por que sedutor? Economia de recursos computacionais. Por que perigoso? O estado é compartilhado entre todos os usuários. É o escopo padrão de todo serviço no Grails.
- `request` Uma nova instância da classe será criada a cada requisição recebida pela aplicação e existirá enquanto aquela requisição estiver ativa. Repare que é diferente do escopo `prototype`. Um bean `prototype` pode sobreviver por muito tempo após finalizada uma requisição.
- `flash` O objeto existirá durante a requisição corrente e a próxima apenas para um dado usuário.

Qual escopo adotar depende do contexto de uso do seu serviço. Se este não possuir estado algum, `singleton` pode ser uma escolha bastante interessante pois lhe economizará muitos recursos computacionais (instanciar objetos é caro tanto do ponto de vista da CPU quanto de memória, lembre-se disso). Se o serviço requer estado, mas deve viver pouco tempo, talvez `request` ou `prototype` lhe atendam melhor.

O prototype que vira singleton

Fig. 9.1: Atenção redobrada!

Lembre-se que serviços também podem receber a injeção de outros serviços. Sendo assim, muito cuidado com seu serviço que possui escopo `singleton` e que recebe como dependência um bean cujo escopo é `prototype`.

A injeção de dependências em uma instância só ocorre uma vez. Sendo assim, o serviço `prototype` também será injetado uma única vez em seu serviço `singleton`, que será compartilhado por todos os usuários concorrentes do seu sistema.

Bom, após toda esta teoria, uma pergunta fundamental surge: como defino o escopo do meu serviço? Simples: com o atributo estático `scope`, que recebe como valor uma string representando o nome do escopo:

```
class ComunicacaoService {
    // Definindo o escopo prototype
    static scope = "prototype"
}
```

Lembre-se: se não for definido o escopo do seu serviço, este será `singleton` por padrão.

9.3 Transações

Um dos recursos mais interessantes dos serviços é o fato de serem transacionais, *mas o que é uma transação?* Para entendê-las, primeiro você precisa conhecer a ideia de *unidade de trabalho* (*unity of work*), ou seja, é a tentativa

de se conseguir tratar um conjunto de tarefas que visam um objetivo final como se fosse uma única ação. O exemplo mais popular é o de movimentações bancárias que repetirei aqui. Imagine a seguinte movimentação:

1) Usuário efetua uma compra na internet usando seu cartão de crédito em um site.

2) O site envia uma mensagem à operadora de cartão de crédito informando a compra.

3) O banco verifica se o usuário pode fazer a compra e avisa ao site se a compra pode ou não ser feita.

4) Aprovada a compra, o estoque do site será decrementado e o pedido processado.

Ou todas as quatro operações são executadas com sucesso ou nenhuma é. Este conjunto de atividades forma uma *unidade de trabalho*. Finalizada a execução da transação, temos nosso banco de dados (ou bancos de dados no caso de transações distribuídas) todos em um estado consistente.

Transações ACID

Transações em bancos de dados relacionais normalmente são caracterizadas pelo acrônimo ACID:

- **Atômicas** o conjunto de tarefas é visto como sendo única unidade de trabalho.
- **Consistente** o estado do banco de dados é mantido consistente caso a transação seja finalizada com sucesso ou não.
- **Isoladas** múltiplas transações ocorrem ao mesmo tempo contra o banco de dados, mas uma não interfere no funcionamento da outra.
- **Duráveis** finalizada com sucesso a transação, os dados encontram-se persistidos e mantidos no banco de dados com segurança. Caso ocorra alguma queda de servidor ou qualquer outro erro fatal, quando o SGBD voltar a operar normalmente, os dados alterados durante a transação continuarão armazenados em disco e consistentes.

É importante mencionar que no caso de bases de dados NoSQL nem sempre podemos contar com transações deste tipo pois a obtenção desta característica irá depender do fornecedor do seu SGBD.

Serviços normalmente lidam com uma série de classes de domínio, alterando normalmente mais de um registro em nosso banco de dados. Ter a segurança de que todos os dados estarão consistentes ao final de uma transação é mais uma razão pela qual implementar serviços é uma **excelente** prática.

Voltando ao nosso serviço `ComunicacaoService`, você deve se lembrar que a classe criada pelo comando `create-service` veio com a anotação `@Transactional` sobre a declaração da classe. Isso quer dizer que todos os métodos presentes em nosso serviço serão executados dentro de transações.

```
import grails.transaction.Transactional

@Transactional
class ComunicacaoService {
}
```

No entanto, a presença desta anotação incluída pelo comando `create-service` é uma novidade da versão 2.3.0 do Grails. Se nosso serviço fosse implementado tal como no exemplo a seguir:

```
class ComunicacaoService {

}
```

todos os seus métodos **por padrão** iniciariam uma nova transação quando fossem executados. A anotação `@Transactional` tem como objetivo principal nos fornecer uma configuração mais fina a respeito de quais métodos devem ou não ser executados dentro de uma transação.

Se quiser desativar todas as transações em um serviço, basta incluir o atributo estático `transactional` com valor igual a `false`:

```
class ComunicacaoService {
    static transactional = false
}
```

Caso seja de seu interesse que algum dos métodos não seja executado dentro de uma transação, basta usar a anotação `@NonTransactional`, o que permite uma configuração mais fina sobre o modo como transações são iniciadas em nossos serviços.

```
import grails.transaction.Transactional
import grails.transaction.NonTransactional
@Transactional
class ComunicacaoService {
    // Este método não será executado
    // em uma transação
    @NonTransactional
    def naoTransacional() {
```

```
    }
}
```

Quando transações não ocorrem

Fig. 9.2: Atenção redobrada!

Transações só ocorrem em serviços injetados em nossos objetos, ou seja, só funcionam em instâncias que são gerenciadas pelo Spring. Sendo assim, se você instanciar manualmente uma classe de serviço o contexto transacional não será ativado.

O código a seguir não executa transacionalmente:

```
new ComunicacaoService().enviarMensagem(item, "kico@itexto.com.br", "Oi!"
```

Rollback

Finalizada a execução com sucesso de um método presente em seu serviço, os dados se encontrarão persistidos corretamente em seu banco de dados. É o que chamamos de operação de `commit`. No entanto, tão importante quanto saber quando o `commit` ocorre (acabei de lhe contar) é entender quando o `rollback`, ou seja, o cancelamento das operações que compõem sua transação ocorre.

O `rollback` ocorrerá de forma automática sempre que uma exceção do tipo `runtime` ocorrer durante a execução de um método. Esse tipo de exceção é aquela que consiste em uma subclasse de `java.lang.RuntimeException` ou então a própria. Observe o exemplo a seguir:

```
class ComunicacaoService {
    def exemploRollback() {
        def itens = concot.Item.list()
        // Código suicida
        for (item in itens) {
            item.delete()
        }
        throw new RuntimeException("Volte para o estado anterior")
    }
}
```

Esse código suicida irá apagar todos os itens cadastrados no banco de dados (da maneira menos performática inclusive, diga-se de passagem), mas ao final é disparada uma exceção do tipo `java.lang.RuntimeException`. Com isso, a operação de `rollback` é chamada de forma automática e nenhuma exclusão se concretizou em nosso banco de dados.

Fig. 9.3: Atenção redobrada!

É importantíssimo salientar que **o `rollback` automático só ocorrerá com exceções do tipo `runtime`**. A lógica por trás desta convenção (que é do Spring, não do Grails) é bastante interessante. Exceções deste tipo ocorrem quando algo catastrófico ocorreu com o sistema como, por exemplo, a queda de algum servidor ou algum erro gravíssimo que não pode ser contornado pelo programador.

Sendo assim, o leitor deve estar se perguntando agora o seguinte: como eu executo uma operação de `rollback` programaticamente? Usando o método `withTransaction`, que é injetado pelo Grails em toda classe de domínio e que recebe como parâmetro uma closure. Observe o exemplo:

```
class ComunicacaoService {
    def exemploRollback() {
        Item.withTransaction { status ->
```

```
            Item.list().each {
                it.delete()
            }
            status.setRollbackOnly()
        }
    }
}
```

A closure passada como parâmetro para o método `withTransaction` recebe um único parâmetro que chamamos `status`. Trata-se de um objeto do tipo `org.springframework.transaction.TransactionStatus` (observe a manifestação do Spring no Grails), cujo método `setRollbackOnly` executa a operação de `rollback` da transação.

Esta classe possui mais algumas funções cuja menção é importante:

- `isCompleted` retorna `true` caso a transação tenha sido concluída.
- `isRollbackOnly` retorna `true` caso tenha sido executada a operação de `rollback`.

Atenção à anotação @Transactional

Fig. 9.4: Atenção redobrada!

Se a anotação `@Transactional` for aplicada a apenas um método de nossa classe, apenas aquele método será transacional.

Para que todos os métodos sejam transacionais por padrão e você queira usar `@Transactional`, aplique a anotação primeiro sobre a classe.

Isolamento

Você pode definir o isolamento da sua transação graças à anotação `@Transactional` e seu atributo `isolation`. Este recebe como valor uma das constantes definidas no enum `org.springframework.transaction.annotation.Isolation`. Primeiro devemos falar brevemente sobre as opções que este enum nos oferece para em seguida tecer alguns comentários. Como verá, a questão do isolamento é muito mais complicada do que aparenta.

- `READ_COMMITED` Sua transação só terá acesso a registros no banco de dados que tenham sido comitados por outras transações. Permite no entanto a ocorrência de leituras não repetitivas.
- `READ_UNCOMMITED` Sua transação terá acesso a registros no banco de dados que ainda não foram comitados por outras transações. É o nível de isolamento que lhe permite melhor desempenho, mas também o mais perigoso pois seu processamento pode levar em consideração registros que não mais existirão após a finalização da transação. Possibilita leituras sujas (*dirty reads*).
- `REPEATABLE_READ` Garante que se você for ler um registro no banco de dados repetidas vezes durante a transação, este sempre retornará com o mesmo valor; mas não consegue evitar leituras fantasma (*phantom reads*).
- `SERIALIZABLE` Garante o maior isolamento possível, no entanto é a configuração mais cara do ponto de vista computacional, pois irá bloquear os registros do seu banco de dados e pode impactar fortemente o desempenho do seu sistema.
- `DEFAULT` Padrão adotado pelo Grails. Suas transações irão adotar as configurações padrão do seu SGBD. Cada um possui um valor padrão. MySQL, por exemplo, adota `REPEATABLE_READS` como padrão.

No exemplo a seguir configuramos a transação do método `enviarMensagem` para o isolamento do tipo `READ_UNCOMMITED`.

```
import grails.transaction.Transactional
import org.springframework.transaction.annotation.Isolation
@Transactional
class ComunicacaoService {
    @Transactional
      (isolation=Isolation.READ_UNCOMMITED)
    def enviarMensagem() {
        // conteudo ignorado
    }
    // Neste método o isolamento padrão será DEFAULT
    def outroMetodo() {

    }
}
```

Na descrição dos valores possíveis para o atributo `isolation` mencionei algumas situações que no padrão SQL/92 são chamados de *fenômenos de leitura* (*read phenomena*) [24] envolvendo o isolamento de transações. É importante que você os conheça para que, com isso, possa selecionar qual a melhor opção para o seu caso.

O primeiro destes fenômenos se chama *leitura suja* (*dirty reads*) e ocorre quando uma transação possui permissão de leitura de registros que foram modificados mas ainda não comitados por outras transações concorrentes. A melhor maneira de entender este fenômeno é através de um exemplo simples envolvendo duas transações: T1 e T2.

```
// T1 Executa a consulta abaixo na tabela nome
select nome from item where id = 1
// obtém o valor 'Motor elétrico'

//Na sequência, T2 executa uma operação de alteração
//no mesmo registro
update item set nome = 'Motor' where id = 1

//T1 executa uma segunda consulta contra o mesmo registro
select nome from item where id = 1
// obtém o valor 'Motor'
```

Como pode ser visto, na segunda leitura T1 perdeu o isolamento, pois o valor foi modificado por T2 executando em paralelo. `READ_UNCOMMITED` é a opção que nos fornece melhor desempenho pois o SGBD não precisa se preocupar tanto com o isolamento das transações, mas você irá encontrar situações como est.

O segundo destes fenômenos se chama *leitura não repetitiva* (*non repeatable reads*). Novamente, o que ocorre é a questão da concorrência envolvendo nossas já conhecidas transações T1 e T2. A diferença está no fato de que uma delas será finalizada antes da outra. Nada melhor que outro bom exemplo para ilustrar a situação.

```
//Nossas duas transações já encontram-se iniciadas
//T1 lê o nome do item
select nome from item where id = 1
// obtém o valor 'Motor elétrico'

//T2 altera o valor do item e em seguida executa commit
update item set nome = 'Motor' where id = 1
commit;

//T1 lê novamente o nome do item
select nome fro mitem where id = 1
//obtém o valor 'Motor'
```

A leitura não repetitiva ocorre quando para uma mesma consulta é executada duas ou mais vezes em uma mesma transação e os valores obtidos são diferentes porque uma outra transação paralela alterou os dados e os comitou antes de sua finalização. Observe a diferença em relação à leitura suja: neste caso, estamos lendo dados que foram COMITADOS novamente, mas cujo valor é diferente na segunda leitura.

Nossa terceira situação se chama *leitura fantasma* (*phantom reads*) e trata-se de um caso especial da *leitura não repetitiva*. Ocorre quando uma transação paralela insere registros no banco de dados e acidentalmente os obtemos em uma segunda consulta. Mais um exemplo prático envolvendo nossas amigas T1 e T2:

```
//T1 busca todas as cotações para um item
//com valor entre 10 e 30
```

```
select * from cotacao where item_id = 1 and valor between 10 and 30
//obtém um conjunto de 2 registros

//T2 insere um novo registro e finaliza antes de T1
insert into cotacao (item_id, valor) values (1, 12)
commit

//T1 executa a mesma consulta novamente
select * from cotacao where item_id = 1 and valor between 10 and 30
//obtém um conjunto de 3 registros
```

Este terceiro registro é o que chamei de "registro fantasma". É como se ele simplesmente tivesse surgido "do nada", concorda? Esta é uma situação inclusive bastante perigosa, pois em casos nos quais um grande número de registros é inserido concorrentemente à sua transação, ela pode simplesmente demorar em excesso para ser finalizada (não param de chegar novos itens!). `NON_REPEATABLE_READS` resolve justamente este tipo de problema.

Bom Kico, agora você me confundiu todo. Qual nível de isolamento devo usar? A resposta é simples: se você tiver um DBA ao seu alcance ou você tiver um conhecimento mais profundo sobre seu banco de dados, pergunte a ele; se não souber, mantenha a configuração padrão do Grails. O importante é que você saiba neste momento que não existe bala de prata quando o assunto é isolamento de transações, mas que é possível pelo menos mudar esta configuração caso se mostre necessário, certo?

Transações somente leitura

Sua transação também pode informar ao SGBD que está apenas fazendo leitura dos dados, o que lhe fornece a oportunidade de aplicar bloqueios menos agressivos em suas estruturas e com isto propiciar um melhor desempenho para seus clientes.

Usando a anotação `@Transactional`, basta definir o atributo `readOnly` com valor `true` como no exemplo a seguir:

```
@Transactional(read=true)
def metodoSomenteLeitura() {

}
```

O leitor mais curioso deve ter percebido que esta anotação encontra-se aplicada em alguns controladores gerados pelo *scaffolding*, tal como no exemplo a seguir:

```
@Transactional(readOnly = true)
class CotacaoController {

}
```

E neste momento deve estar se perguntando: *controladores podem iniciar uma transação?*. Resposta rápida: sim! Isso porque são *beans* assim como os serviços, portanto, podem se beneficiar de todos os benefícios deles. No entanto, isso não quer dizer que você deva escrever sua lógica de negócio neles, mas há situações nas quais não vale a pena escrever um serviço. Pense em uma action, por exemplo, que apenas lista itens para seu usuário final. Realmente precisaríamos de um serviço ali?

Propagação

Finalmente, o terceiro conceito sobre transações que deve fazer parte do seu cinto de utilidades: como uma transação se propaga? É melhor explicar este conceito com algum código: imagine os dois métodos presentes em um serviço imaginário:

```
class ImaginarioService {
    def metodo1() {
        // executa uma série de ações
        metodo2()
    }

    def metodo2() {
        //executa mais uma série de ações
    }
}
```

Por padrão, todo método em um serviço é transacional. No entanto, vemos aqui que `metodo1` faz uma chamada a `metodo2`. O que ocorre com a transação iniciada em `metodo1`? Ela é finalizada ou continua a mesma em `metodo2`? Ou é finalizada em `metodo1` e uma nova é criada em `metodo2`?

A transação que foi iniciada em `metodo1` será finalizada em `metodo2`. No entanto, você pode customizar este comportamento alterando o atributo `propagation` da anotação `@Transactional`. Novamente, vamos ver quais são os valores disponíveis, que são as constantes definidas no enum `org.springframework.transaction.annotation.Propagation`.

- `REQUIRED` Quando o método for invocado, uma nova transação será iniciada. Caso uma transação já esteja em execução, esta é mantida.
- `SUPPORTS` Se uma transação já tiver sido iniciada, ela será mantida durante a execução do método. Caso não haja nenhuma transação em execução, o método funciona da mesma forma.
- `MANDATORY` O método só é executado se uma transação estiver em execução. Tentar executá-los em uma irá disparar uma exceção em tempo de execução.
- `REQUIRES_NEW` O método ao ser executado **sempre** irá criar uma nova transação. Se ele for chamado e uma transação já estiver em execução, esta será paralisada até sua finalização.
- `NOT_SUPPORTED` Se houver uma transação em execução, irá paralisá-la até sua finalização. Internamente executa sem que qualquer transação seja iniciada.
- `NEVER` Executa de forma não transacional. Se uma transação estiver em execução, uma exceção será disparada.
- `NESTED` Se já houver uma transação iniciada, esta é paralisada. Enquanto isto, o novo método criará uma nova transação independente, a executará e, em seguida, retornará o controle para a anterior.

No exemplo a seguir mudamos a propagação da transação para `NEVER`.

```
import grails.transaction.Transactional
import org.springframework.transaction.annotation.Propagation
@Transactional
class PropagacaoService {
```

```
    @Transactional(propagation=Propagation.NEVER)
    def transacaoNever() {

    }

}
```

9.4 Falando de Spring

Neste capítulo, pudemos ver que o grande ator oculto por trás do Grails é o Spring. Tiramos proveito de diversas das suas funcionalidades como a injeção de dependências, inversão de controle, transacionalidade, nossos controladores, data binding e diversos outros pontos. E sabem o que é mais interessante? Você pode acessar e modificar as configurações do Spring ou mesmo acessar o seu container de injeção de dependências diretamente, sabia? Nesta seção, iremos mostrar como tirar proveito destes recursos.

Adicionando seus próprios beans

Na maior parte das vezes, controladores, classes de domínio e serviço irão atender todas as suas necessidades, mas e se você tiver algum código legal com funcionalidades que seriam interessantes de serem reaproveitadas em nosso sistema?

No *ConCot* temos uma classe, originalmente implementada em Java usada para calcular impostos. Chama-se `CalculoImpostos.java` e sua implementação pode ser vista a seguir:

```
public class CalculoImpostos {

    /** Tarifa a ser aplicada */
    private double tarifa;

    public double getTarifa() {
        return tarifa;
    }
```

```
    public void setTarifa(double valor) {
        this.tarifa = valor;
    }

    public BigDecimal calcularTarifa(double valor) {
        // Quem dera no mundo real ser tão
        // fácil assim calcular impostos!
        return valor * tarifa;
    }

}
```

Observe que ela possui uma propriedade chamada `tarifa`, que usaremos na sua transformação em bean. Como uma classe vira um bean? Notificando o Spring de que ela existe. A maneira mais simples é modificando o arquivo `grails-app/conf/spring/resources.groovy` que originalmente é exatamente como o exposto a seguir:

```
beans = {
}
```

Para incluir nosso *bean* `CalculoTarifa`, basta modificá-lo para que fique desta forma:

```
beans = {
    calculoTarifa(concot.CalculoImpostos) {
        tarifa = 0.5
    }
}
```

O nome do nosso bean será `calculoTarifa`, que irá instanciar um objeto do tipo `concot.CalculoImpostos`, cuja implementação encontra-se no arquivo `src/java/concot/CalculoImpostos.java`. Observe que, logo em seguida, definimos o valor das propriedades do nosso *bean* : `tarifa` receberá o valor `0.5`.

Na sequência, basta injetar este bean onde se faça necessário, por exemplo, no serviço a seguir. Tudo o que você precisa fazer é declarar um atributo em sua classe cujo nome corresponda ao do bean que acabamos de declarar:

```
class TributoService {

    // Bean injetado
    def calculoTarifa

    def calcular(double valor) {
        // basta usar o bean
        calculoTarifa.calcularTarifa(valor)
    }
}
```

Acessando o container do Spring diretamente

Finalmente, você também pode acessar o container do Spring diretamente através da sua aplicação Grails. Isso é bastante útil para quando, por exemplo, queremos evitar problemas como o do `prototype que se torna singleton` que mencionei agora há pouco.

Há duas maneiras de se conseguir isso. A primeira é modificar nossa classe de serviço para que esta implemente a interface `org.springframework.context.ApplicationContextAware` do Spring tal como no exemplo a seguir:

```
import org.springframework.context.ApplicationContextAware
import org.springframework.context.ApplicationContext

class SpringAwareService implements ApplicationContextAware {

    //O contexto do Spring
    private ApplicationContextAware contextoSpring

    void setApplicationContext(ApplicationContext ctx) {
        this.contextoSpring = ctx
    }

}
```

Ao implementarmos esta interface, as instâncias do nosso serviço irão ter invocado o método `setContextAware` após terem todas as suas de-

pendências injetadas, permitindo acessar o contexto de aplicação do Spring, representado pelo parâmetro `ctx` que foi exposto.

A partir deste ponto você pode acessar livremente o contexto e obter o bean que você quiser pelo nome ou pela classe. Lembra aquele `prototype` que virava `singleton`? Veja como podemos evitar o problema:

```
class SpringAwareService implements ApplicationContextAware {

    //O contexto do Spring
    private ApplicationContextAware contextoSpring

    void setApplicationContext(ApplicationContext ctx) {
        this.contextoSpring = ctx
    }

    def obtenhaBeanPrototype() {
        contextoSpring.getBean("beanPrototype")
    }

    def usePrototype() {
        obtenhaBeanPrototype().executeAlgo()
    }

}
```

A função `getBean` do contexto do Spring retorna um bean pelo seu nome. Ao chamá-lo, o Spring irá nos retornar um novo *bean* com escopo prototype sempre. Em seguida, nós simplesmente o usamos, tal como é feito na função `usePrototype` e este será descartado pelo próprio *garbage collector* da JVM após algum tempo, pois não foi associado a nenhum atributo da nossa classe.

Há também uma variante do método `getBean` do objeto `ApplicationContext` que recebe como parâmetro uma classe e lhe retornará o bean do mesmo tipo.

Tirando proveito do ciclo de vida dos beans

O container de IoC do Spring lida com o ciclo de vida dos beans, ou seja,

ele que sabe quando um bean "nasce" e "morre". Você pode tirar proveito disso, sabia? Basta implementar algumas interfaces bem simples do Spring.

Se quiser, por exemplo, que seu bean execute algum código de inicialização após ter tido todas as suas dependências injetadas, basta implementar a interface `org.springframework.beans.factory.InitializingBean`:

```
import org.springframework.beans.factory.InitializingBean

class InicializadorService implements InitializingBean {

    /*
        Aqui entra seu código de inicialização
    */
    void afterPropertiesSet() {

    }
}
```

Implementando-a, seu serviço será construído da seguinte maneira:

1) O Spring irá criar uma nova instância da classe.

2) Serão injetadas todas as dependências nesta instância.

3) O método `afterPropertiesSet` é executado.

4) O bean é disponibilizado para ser injetado em outro ou usado por você.

Esta é uma funcionalidade bastante útil quando você precisa, por exemplo, executar algum preparo inicial no seu bean antes que este seja disponibilizado para o sistema. Se seu serviço precisa de dados imutáveis presentes no banco de dados, este é um bom momento para que os mesmos sejam carregados.

Outra vantagem é evitar erros de concorrência. Imagine que duas *threads* tentem acessar o bean ao mesmo tempo e, neste momento, ele precise carregar estas informações a partir do banco de dados por não ter sido previamente preparado. Ele poderia acidentalmente executar o mesmo código de consulta

DUAS vezes, ou mesmo você poderia se deparar com erros de concorrência que são dificilmente detectados ou resolvidos.

Outro ponto do ciclo de vida de que você pode tirar proveito é o momento em que o bean é descartado pelo container. Novamente, basta implementar outra interface do Spring: `org.springframework.beans.factory.DisposableBean`. O método `destroy` será executado quando o bean for descartado. A seguir você pode ver um exemplo rápido de sua aplicação:

```
org.springframework.beans.factory.DisposableBean

class LimpezaBean implements DisposableBean {
    void destroy() {
        // executa a limpeza
    }
}
```

Isso é muito útil para aquele código que "arruma a casa" quando seu bean não é mais necessário. Pense em ações como fechamento de conexões com o banco de dados ou outro serviço qualquer, por exemplo, que você mantenha abertas durante a execução do bean devido a alguma razão de otimização. Outro bom uso é a exclusão de arquivos temporários.

Cuidado com o prototype

Fig. 9.5: Atenção redobrada!

Beans do tipo `prototype` saem do domínio do container do Spring assim que são gerados. Dessa forma, o Spring apenas conseguirá executar o método `afterPropertiesSet`, e não `destroy`.

Tenha isso em mente ao usar a interface `DisposableBean`.

Capítulo 10

Testes

Grails é um framework que incentiva seus usuários (programadores) o tempo inteiro a escreverem testes. Provavelmente, você já observou que ao executarmos comandos como `create-controller` ou `create-tag-lib` também são gerados esqueletos para testes na pasta `test/unit` do projeto. Chegou a hora de explorar este conteúdo.

10.1 Antes um pouco de metodologia: TDD e BDD

Um assunto que até hoje gera polêmica são as práticas de TDD (*Test Driven Development*) e BDD (*Behaviour Driven Development*). Ouvimos sempre objeções como *"agora vou gastar muito tempo escrevendo testes*

, *"minha produtividade (ou da minha equipe) irá baixar*

e diversos outros. Recentemente, David Heinemeier Hansson (criador do Ruby on Rails) levantou novamente esta questão ao publicar em seu blog o texto *TDD is dead. Long live testing* [12] que curiosamente é bastante citado mas não lido (quando é lido) com a atenção que merece.

Na minha experiência como consultor, observo que a adoção destas práticas nas empresas em que atuo gera resultados bastante interessantes: desenvolvedores mais confiantes, acúmulo menor de horas extras, código mais fácil de ser mantido e, não raro, um aumento até mesmo da autoestima da equipe como um todo. Sei que quando estamos começando é difícil acreditar no que estas práticas (TDD e BDD) prometem, mas garanto que se você se esforçar irá entender claramente seu valor. E sabe de uma coisa? Vamos aplicá-las neste capítulo, mas antes vou falar um pouco a seu respeito.

TDD em cinco minutos

A prática do TDD (*Test Driven Development*) pode ser resumida de uma forma bastante rudimentar em uma frase: *escreva seus testes antes do código*. É na realidade a aplicação de um modelo de desenvolvimento baseado em rápidos e pequenos ciclos.

A cada nova funcionalidade, primeiro o desenvolvedor escreve um teste que descreve seu comportamento esperado usando sua ferramenta favorita (no caso do Grails e deste livro, vamos adotar o Spock). Somente aí o código-fonte é escrito pela primeira vez e será executado repetidas vezes contra o teste até que este seja finalmente executado com sucesso.

Passada a primeira fase, entra a segunda na qual você deverá, se necessário, refatorar o código para que fique de acordo com as melhores práticas de desenvolvimento ou mesmo os padrões internos adotados pelo seu time. E como saber se suas mudanças não irão estragar o que acabou de fazer? Executando os testes contra elas.

Sei que soa muito estranho e contraprodutivo dizer que escrever os testes **antes** do código pode aumentar a sua produtividade, mas o que observo é que quando nos exercitamos nesta direção acabamos por refletir melhor a respeito **do objetivo final**. E com isso você acaba conhecendo melhor não só o que deve ser feito, mas também **quando** está pronto (quando os testes passam).

Mais do que isto, também noto que os testes são excelentes para se detectar código em estado de decomposição. Como? Simples, se está difícil escrever um teste que descreva como aquele código deveria se comportar, é sinal de que há ou problemas no entendimento ou no modo como escrevemos aquele código.

Claro, o desenvolvedor deve tomar cuidado também **com o que testa**. Não vale a pena testar o uso de código que todos sabemos funcionar como, por exemplo, o seu framework de persistência, tal como no exemplo a seguir:

```
def salvar(Cotacao cotacao) {
    cotacao.save()
}
```

Nós já sabemos como o GORM funciona. Testar o método `save` não irá agregar nada ao seu projeto, a não ser que você esteja explorando o funcionamento da API ou reportando um bug à equipe de desenvolvimento do Grails.

É como testar um *getter* ou *setter* Java: associação de valores é algo que obrigatoriamente a linguagem de programação deve nos fornecer para que possa funcionar minimamente. Então, o que vale a pena testar? A lógica pela qual **você** é responsável, e não seu framework, linguagem de programação ou biblioteca.

Normalmente, as críticas envolvendo produtividade na escrita de testes nascem deste ponto: a escrita de testes que na realidade não agregam, mas apenas complicam sua vida.

Além da vantagem de facilitar o processo de design, outra característica interessante dos testes é o fato de estes acabarem se tornando uma espécie de "documentação viva" do seu sistema, como uma especificação formal e que acaba por aumentar o sentimento de segurança na sua equipe. Vocês sabem agora quando suas alterações no sistema introduzem novos bugs: os testes lhe contam!

E o BDD?

Já vi a prática do TDD operar milagres em diversos projetos, no entanto há uma pequena limitação: seu foco tende a ser os desenvolvedores. A prá-

tica do BDD [20] (*Behaviour Driven Development*) pode ser vista como uma evolução do TDD, pois iremos trazer para o nosso lado não apenas testadores profissionais, mas sim **todos** aqueles que possuem interesse no sistema a ser desenvolvido.

O termo BDD foi introduzido por Dan North [21] a partir da sua experiência como instrutor de TDD. Observando as suas próprias dificuldades, assim como a dos seus alunos [21], North acabou por criar este conceito que vai muito além dos testes. Trata-se de uma nova forma de desenvolvimento.

Dentre as inovações trazidas pelo BDD está o conceito de *linguagem compartilhada* (*ubiquitous language*), que nada mais é que a busca por um vocabulário comum, compreendido por todos os interessados no sistema, que permita satisfazer alguns objetivos:

- Garantir que todos se entendam da melhor maneira possível. Desenvolvedores, analistas de requisitos e, principalmente o cliente final deverão todos falar o mesmo idioma.
- Construir o conceito de **pronto**, ou seja, **todos** saberão com precisão quando determinada funcionalidade encontra-se inteiramente implementada e pronta para o ambiente de produção.

Mencionamos aqui o conceito de BDD pelo fato de no Grails usarmos como framework de testes o Spock, que tem como foco auxiliar a adoção desta prática. Como veremos, seu estilo de escrita é bastante diferente, possibilitando-nos escrever testes de tal modo que todos os envolvidos no projeto poderão entendê-los e, ainda mais importante, adotá-los como parte importante da documentação do sistema.

A propósito, haverá também algumas mudanças em nosso vocabulário. Ao pensarmos em BDD, trocaremos algumas palavras. Usaremos os termos "comportamento" (*behaviour*) e especificação em vez de "testes". Para saber mais sobre o porquê desta mudança, recomendo que você leia o texto *Introducing BDD* do próprio Dan North [21].

10.2 Nossas primeiras especificações

Em uma bela manhã Guto procurou Daniel pedindo um novo recurso para o *ConCot*.

Daniel, aconteceu algo muito chato aqui. Um dos nossos clientes recebeu nossa lista de cotações para seu projeto e achou que diversos dos itens que incluímos lá eram exageradamente caros para ele. Como se trata de uma pessoa muito "sensível", achou que estávamos esnobando-o e nos enviou um e-mail criticando nosso trabalho.

Ele é doido?

Não, apenas pão-duro e chato. Então, estava pensando, será que nós não poderíamos criar "listas de cotações" para nossos clientes?

Como assim?

Explico melhor: imagine que tenhamos um cliente X. Criaríamos uma lista de cotações exclusiva para ele. Nela, poderíamos incluir limites de valor. Por exemplo: "o valor máximo de um britador é R$ 10.000,00" para esta listagem. Com isso, o sistema só permitiria incluir cotações com valor de no mínimo R$ 10.000,00. O que acha?

Hmm... entendi. E tem limite mínimo também?

Não havia pensado nisto, mas é uma boa ideia! Também pode ter um valor mínimo. Tipo: no mínimo R$ 100,00 para aquele item. O que acha, é viável?

Acho que sim, Guto. Olhando aqui, teríamos de incluir mais duas entidades no *ConCot*. Uma para representar a lista e outra para colocar os limites de valor, certo?

Errado: tem de ter também uma terceira entidade que represente a cotação incluída na lista. De preferência com a data em que foi inserida. Pode ser?

Uai, pode!

Ah, e outra coisa Daniel: tem de ser possível que o valor máximo e mínimo do item possam ser opcionais, mas que pelo menos um exista.

Como assim?

Por exemplo: um item pode ter valor mínimo e máximo, ou apenas o valor mínimo ou apenas o valor máximo, entendeu?

Entendeu!

Com base nesta conversa Daniel adicionou as novas entidades no sistema e terminou com um modelo de entidades tal como o exposto na imagem a seguir:

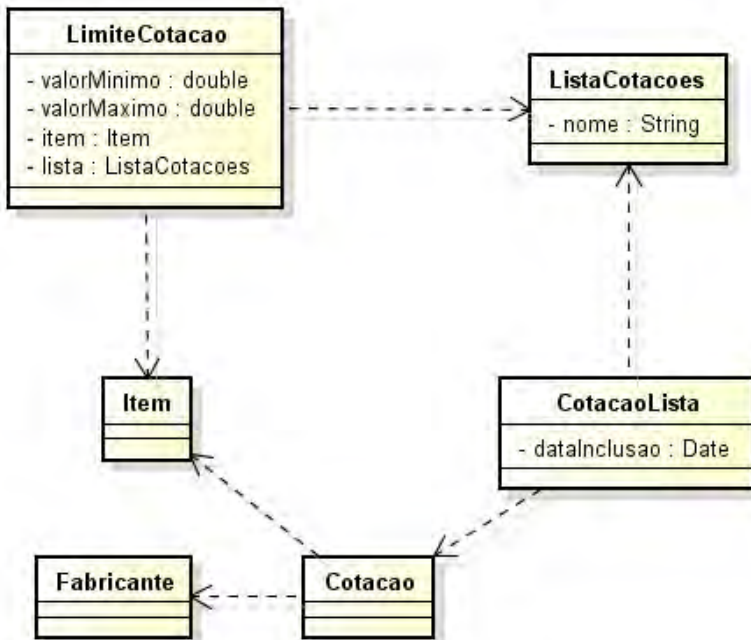



Fig. 10.1: Diagrama de classes do Concot

E implementou rapidamente todas as classes de domínio:

```
class ListaCotacoes {

    String nome

    static constraints = {
        nome nullable:false, blank:false, maxSize:128, unique:true
    }
}
```

```
package concot

class LimiteCotacao {


    BigDecimal valorMinimo
    BigDecimal valorMaximo

    static belongsTo = [lista:ListaCotacoes, item:Item, moeda:Moeda]

    static constraints = {
        valorMinimo: nullable:true
        valorMaximo: nullable:true
    }
}

class CotacaoLista {

    Date dataInclusao

    static belongsTo = [
        lista: ListaCotacoes,
        cotacao: Cotacao
    ]

    static constraints = {
    }
}
```

Daniel já conhecia o Spock, que é o framework de testes adotados pelo Grails como padrão desde a versão 2.2 e também já estava estudando BDD. Sendo assim, com base na conversa que teve com Guto achou que seria uma boa ideia colocá-los em prática.

Começou percebendo que o mais importante não seriam as telas de cadastro, mas sim a lógica de negócio. Nada mais interessante do que começar pela implementação de um serviço que validasse a inclusão de novas cotações em uma lista. Sendo assim, executou o comando `create-service`

ListaCotacoes na interface de linha de comando do Grails e com isso já criou também o arquivo de especificação do Spock que usaria para primeiro escrever seus testes (quer dizer, especificações!).

O arquivo ListaCotacoesServiceSpec foi gerado na pasta test/unit/concot, o que indica se tratar de um teste unitário, e seu conteúdo original já nos mostra com o que Spock se parece:

```
package concot

import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(ListaCotacoesService)
class ListaCotacoesServiceSpec extends Specification {

    def setup() {
    }

    def cleanup() {
    }

    void "test something"() {
    }
}
```

Testes unitários e de integração

Grails nos fornece por padrão dois tipos de testes: unitário e integrado. O objetivo do teste unitário é nos fornecer o ferramental necessário para que possamos verificar o funcionamento de uma funcionalidade de modo isolado, ou seja, sem que precisemos iniciar todo o sistema para isto.

Isso nos permite focar apenas na lógica que estamos implementando naquele trecho do sistema, mas por outro lado, dado que sempre precisamos interagir com outros componentes arquiteturais, por exemplo, a camada de persistência, um novo problema surge. Como fazê-lo? A resposta é simples: criando versões falsas dos mesmos (os *mocks*). Veremos com detalhes neste capítulo como fazer isso.

Testes unitários nos fornecem, portanto, dependências do nosso código que funcionam em "condições ideais de temperatura e pressão", ou seja, elas funcionam exatamente como gostaríamos, o que pode ser um problema, mas nos ajuda a focar melhor naquilo que desejamos verificar.

Já testes integrados resolvem o problema da "condição ideal de temperatura e pressão" ao nos disponibilizar o sistema completo. Você não precisa mais de mocks, pois estará usando o próprio sistema para isso. São fundamentais para que vejamos qual será o comportamento do sistema quando for para produção, fornecendo-nos um ambiente que é o mais similar possível a este.

Se você já trabalhou com testes usando frameworks como jUnit deve estar achando muito estranha a declaração de um dos métodos:

```
void "test something"() {

}
```

Na realidade, se nunca trabalhou com Groovy ou nunca usou nenhum framework BDD como, por exemplo, Cucumber, também deve ter estranhado bastante. O que temos aqui é a aplicação de um dos princípios adotados por Dan North: *"os nomes dos testes devem ser sentenças*

[21].

Spock nos incentiva a escrever testes cujo nome não seja uma função "padrão" como `testIncluirCotacao`, mas sim frases que possam ser compreendidas, além de por programadores, também pela equipe de negócio, como Guto, por exemplo. Sendo assim, antes de sequer escrever o nosso primeiro método em `ListaCotacoesService`, Daniel já sapeca o primeiro comportamento esperado: um simples teste de sanidade.

```
void "verificando a sanidade do serviço para uma cotação nula"() {
  when:"quando"
      def cotacao = null
      def lista = new ListaCotacoes()
      def data = new Date()
  then:"então..."
      service.incluirCotacao(lista, cotacao, data) == null
}
```

No jargão do Spock, não temos testes, mas sim comportamentos. Este foi o primeiro comportamento a ser definido por Daniel. O que fazer quando passo uma cotação nula para nosso serviço? Neste primeiro momento, Daniel também começou a pensar em como seria a assinatura do método responsável por incluir uma cotação em uma lista. Algo que deve receber três parâmetros: a lista na qual será feita a inclusão, a cotação e uma data.

Daniel também achou interessante definir aqui qual o tipo de retorno desta função: uma instância de `CotacaoLista` já persistida, informando assim ao desenvolvedor que o item pode ser incluído com sucesso na lista de cotações. Mas o que são estas instruções `when:` e `then:`.

Elas demarcam blocos de iteração usados pelo Spock. O primeiro, `when:` (*quando*), é usado para definir as condições em cima das quais deveremos verificar um comportamento. É normalmente onde definimos as variáveis que serão usadas em nossa verificação, como no escrito por Daniel em que iremos checar como nosso código se comportará ao receber como valor uma cotação nula.

O próximo bloco, `then:`, demarca aquilo que queremos verificar. Neste caso, deve ser escrita uma expressão booleana por linha. Voltando ao caso do

Daniel, ele quer se certificar de que será retornado o valor nulo quando uma cotação nula for passada como parâmetro para nosso teste.

E aquele `service` ali, de onde vem? Vamos nos lembrar do início do arquivo onde estamos escrevendo nossa especificação:

```
package concot

import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(ListaCotacoesService)
class ListaCotacoesServiceSpec extends Specification {
```

A anotação `@TestFor` recebe como parâmetro o nome da classe de serviço que criamos com o comando `create-service`. Ela instrui o Grails a, no momento em que os testes forem executados, injetar uma instância de `ListaCotacoesService` na classe `ListaCotacoesServiceSpec`.

Esta instância será um atributo cujo nome, para serviços, é `service`. Ela pode ser aplicada a todos os artefatos testados do Grails (serviços, classes de domínio, controladores, bibliotecas de tag) e para cada um destes irá criar um atributo cujo nome corresponda a seu tipo, ou seja, respectivamente `service`, `domain`, `controller`, `tagLib`.

Como todo bom praticante de BDD/TDD, o próximo passo de Daniel é executar os testes, mesmo sabendo que irão falhar (afinal de contas, sequer escrevemos a função `incluirCotacao` ainda!). Para tal, no CLI (*Command Line Interface* Interface de Linha de Comandos) do Grails irá executar o comando `test-app :unit ListaCotacoesService`.

O comando test-app

O comando `test-app` é usado na interface de linha de comandos do Grails para executar os testes (ou especificações!) escritas por nós.

Seu uso é bastantes simples: sem nenhum parâmetro, vai executar todas as especificações, tanto unitárias como integradas.

Se quiser apenas alguma, basta digitar parte do nome, que apenas as que correspondam ao prefixo fornecido serão executadas. Exemplo: `test-app ListaCotacoesService` vai executar o arquivo de testes `ListaCotacoesServiceSpec.groovy`.

Se quiser executar apenas os testes unitários, passe também o valor `:unit` na linha de comando. Apenas os integrados? Use `:integration`.

O resultado esperado é obtido: os testes não irão passar, como pode ser exposto na saída do console exposta a seguir:

```
grails> test-app :unit ListaCotacoesService
| Running without daemon...
.........................................
|Running 2 unit tests...
|Running 2 unit tests... 1 of 2
Failure:  |
verificando a sanidade do servi├º o para uma cota├º├úo nula(concot.ListaCotacoesS
erviceSpec)
 |
groovy.lang.MissingMethodException: No signature of method: concot.ListaCotacoes
Service.incluirCotacao() is applicable for argument types: (concot.ListaCotacoes
, null, java.util.Date) values: [concot.ListaCotacoes : (unsaved), null, Sat Feb
 28 18:42:48 BRT 2015]
        at concot.ListaCotacoesServiceSpec.verificando a sanidade do servi├ºo pa
ra uma cota├º├úo nula(ListaCotacoesServiceSpec.groovy:24)
|Completed 1 unit test, 1 failed in 0m 4s
.Tests FAILED
|
 - view reports in C:\livroGrails\concot\target\test-reports
```

Fig. 10.2: Erro nos testes

É importante mencionar que sempre é gerado também um relatório dos testes nos formatos texto e HTML no diretório `target/test-reports`. A seguir podemos ver um exemplo da saída do relatório atual:

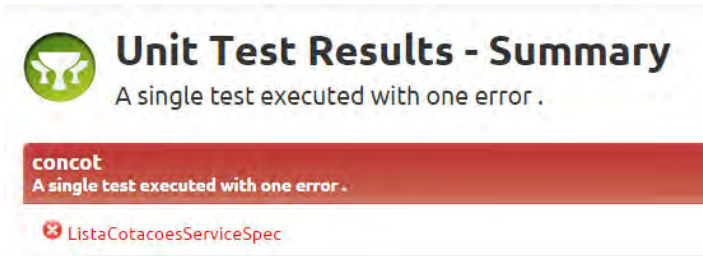



Fig. 10.3: O relatório de testes

Pelo erro que obtivemos nos testes ironicamente sabemos que eles funcionam. Sendo assim, nosso próximo passo é implementar a regra de negócios na classe `ListaCotacoesService` com base no que já temos pré-arranjado em nossa primeira especificação formal, ao menos uma versão que atenda a este primeiro comportamento.

```
class ListaCotacoesService {

/**
    Verifica os limites
*/
  private boolean testeLimites(ListaCotacoes lista, Cotacao cotacao){
    // ainda não implementado neste momento
    true
  }

  def incluirCotacao(ListaCotacoes lista, Cotacao cotacao, Date data){
      if (lista && cotacao && data && testeLimites(lista, cotacao))
    {
          return new CotacaoLista
                        (cotacao:cotacao,
                         lista:lista,
                         dataInclusao:data).save()
      }
      null
  }
}
```

Executando os testes novamente, agora eles passam, mas o trabalho ainda não está completo.

```
Running 2 unit tests...
Running 2 unit tests... 1 of 2
Completed 1 unit test, 0 failed in 0m 4s

Tests PASSED - view reports in C:\livroGrails\concot\target\test-reports
```

Fig. 10.4: Os testes passaram com sucesso!

Não definimos qual comportamento esperamos quando as cotações que submetemos à nossa lista estão dentro ou fora dos limites que nosso cliente definiu. E dado que estamos lidando com testes unitários, de onde iremos obter os dados? Qual banco de dados usar? Grails possui uma versão do GORM feita especificamente para testes unitários que funciona inteiramente em memória.

Tudo o que precisamos fazer é incluir a anotação `grails.test.mixin.Mock` na classe em que definimos nossa especificação informando quais classes de domínio precisamos que sejam mockadas pelo Grails para nós, tal como Daniel fez:

```
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
import spock.lang.Specification

@TestFor(ListaCotacoesService)
// Recebe uma lista com o nome das nossas
// classes de domínio
@Mock([ListaCotacoes, Cotacao, Item,
       Fornecedor, Moeda, Categoria,
       LimiteCotacao, CotacaoLista])
class ListaCotacoesServiceSpec extends Specification {

}
```

O resultado? Podemos escrever nossos comportamentos (*behaviours*) agora como se tivéssemos um banco de dados real por baixo dos panos!

Condições ideais de temperatura e pressão

Fig. 10.5: Atenção redobrada!

Lembre-se que nada substitui um teste integrado. A versão em memória do GORM usada em testes unitários opera em "condições ideais de temperatura e pressão". O desempenho sempre será excelente e você não verá problemas que só encontrará em produção como, por exemplo, erros em índices, problemas de desempenho etc.

A vantagem dos testes unitários está em nos auxiliar no foco da lógica que estamos escrevendo **sem que precisemos nos preocupar com o ambiente de produção** neste momento apenas.

Agora que Daniel pode tirar proveito do GORM em memória, tudo o que precisa fazer é escrever sua segunda especificação.

```
void "uma cotação cujo valor esteja no limite deve poder ser inserida"()
 when:
   def fornecedor = Fornecedor.findOrSaveByNomeAndEmail
      ("Juca", "juca@juca.com")
   def categoria = Categoria.findOrSaveByNome("Equipamento")
   def item = Item.findOrSaveByCategoriaAndNome(categoria, "Motor")
   def moeda = Moeda.findOrSaveByNomeAndSimbolo("Real", 'R$')
   def cotacao =
      Cotacao.findOrSaveByFornecedorAndItemAndMoedaAndValor
          (fornecedor, item, moeda, 100)
   def lista = ListaCotacoes.findOrSaveByNome("Lista de teste")
   def limite = LimiteCotacao
          .findOrSaveByListaAndItemAndMoeda(lista, item, moeda)
   limite.valorMinimo = 10
```

```
    limite.valorMaximo = 1000
    def cotacaoForaDoLimite =  Cotacao.findOrSaveByFornecedorAndItemAndMoeda
       (fornecedor, item, moeda, 1)
    limite.save()
 then:
  service.incluirCotacao(lista, cotacao, new Date()) != null
  service.incluirCotacao(lista, cotacaoForaDoLimite, new Date()) == null
}
```

Uou! Reparou como pudemos tirar total proveito dos finders dinâmicos para gerar automaticamente o estado esperado no banco de dados para que nosso sistema funcione? Novamente, Daniel executará o comando `test-app` apenas para ver os testes falhar e comprovar seu funcionamento. O próximo passo será implementar uma versão de `ListaCotacaoService` que respeite estes comportamentos, o que é feito, gerando uma versão inicial da funcionalidade:

```
class ListaCotacoesService {
 /**
     Verifica os limites
 */
 private boolean testeLimites(ListaCotacoes lista, Cotacao cotacao) {
   def limite = LimiteCotacao.findByItemAndListaAndMoeda
        (cotacao.item, lista, cotacao.moeda)
     if ( limite &&
         ((limite.valorMaximo && cotacao.valor > limite.valorMaximo) ||
          (limite.valorMinimo && cotacao.valor < limite.valorMinimo) ))
         {
           return false
         }

   true
 }

 def incluirCotacao(ListaCotacoes lista, Cotacao cotacao, Date data) {
  if (lista && cotacao && data && testeLimites(lista, cotacao)) {
        return new CotacaoLista
            (cotacao:cotacao, lista:lista, dataInclusao:data).save()
  }
```

```
    null
  }
}
```

Executando os testes novamente, o que temos? Sucesso! Daniel executou o BDD quase que perfeitamente. Sabem qual foi a principal limitação? A ausência de Guto durante a escrita das especificações com Spock. Ao se encontrarem, este erro se tornou evidente.

Olha Guto, aqui estão as especificações que escrevi. O que acha?

Uai Daniel, parece muito legal isto aqui. É a primeira vez que consigo entender seu código!

Não sei se agradeço ou xingo Guto. Mas e aí, o que achou?

Ainda tenho algumas dúvidas: e se eu passar uma lista, uma cotação dentro do limite e a data for nula, retorna falso? E se eu passar uma lista, uma data, e a cotação for inválida, ele vai me retornar nulo? E se eu passar tudo certo menos a cotação, ele vai retornar falso? E se eu passar a data de amanhã? E se for a data de ontem, como fica? E se a lista não tiver nenhuma restrição, qualquer cotação é aceita, certo? E se..? E se...? E se....?

Guto bombardeou Daniel com uma série de exemplos. E esse é o modo como as pessoas fora da sala ou empresas de desenvolvimento pensam. Acredite: eu e você achamos Groovy uma linguagem maravilhosa, mas para seu cliente ou o realmente interessado pelo software que está sendo escrito, ela não passa de algum idioma estrangeiro extremamente feio.

Nem tudo está perdido: Daniel não precisa implementar uma quantidade infinita de especificações por que Spock nos permite escrever verificações baseadas em tabelas!

Daniel começa tirando proveito de uma função incluída por padrão no arquivo `ListaCotacoesServiceSpec` que não vimos até este momento: `setup`. É lá que definimos o estado que poderá ser compartilhado por todos os comportamentos da nossa especificação. E o método `cleanup`? É o que executaremos para limpar qualquer sujeira que tenhamos deixado. É importante mencionar que `setup` sempre é executado antes de todo teste e `cleanup` após todos os testes presentes no mesmo arquivo.

Para facilitar a escrita desse tipo de testes, é necessário que sejam criados alguns atributos na classe `ListaCotacoesServiceSpec`, que devem

ser precedidos pela anotação `@Shared` (você precisará importar o pacote `spock.lang.Shared`). Testes escritos em Spock só podem acessar os atributos da classe na qual são declarados se eles forem estáticos ou precedidos desta anotação. Sendo assim, Daniel ao lado de Guto modifica o início da especificação para que fique tal como exposto a seguir:

```
import spock.lang.Shared

@TestFor(ListaCotacoesService)
@Mock([ListaCotacoes, Cotacao, Item,
       Fornecedor, Moeda, Categoria,
       LimiteCotacao, CotacaoLista])
class ListaCotacoesServiceSpec extends Specification {
    /*
        Lista que contém uma restrição para um item
        "Motor"    cujo valor deve variar entre 10 e 1000 Reais
    */
    @Shared listaTeste
    /*
        Uma lista que não contém restrição alguma
    */
    @Shared listaVazia
    /* Uma cotação para o item "Motor" que vale 100 Reais */
    @Shared cotacaoValida
    /* Uma cotação para o item "Motor" que vale 1 Real */
    @Shared cotacaoInvalida

    def setup() {
        /*
            Código que inicia os atributos acima,
            "persistindo-os" na versão em memória do GORM
            disponibilizada pelo Grails para testes
            unitários
        */
    }
}
```

Logo em seguida, entra nosso novo comportamento, que Daniel escreve ao lado de Guto.

```
void "situações esperadas para a inclusão de cotações na lista"() {
 expect:"o que esperamos"
  resultado == (service.incluirCotacao(lista, cotacao, data) != null)
 where:"alguns exemplos"
  lista      |   cotacao        |   data      |    resultado
  listaTeste | cotacaoValida    | new Date()  |      true
  listaTeste | cotacaoValida    |  null       |      false
  listaTeste | cotacaoInvalida  | new Date()  |      false
  null       | cotacaoValida    | new Date()  |      false
  listaVazia | cotacaoInvalida  | new Date()  |      true
  listaVazia | cotacaoValida    | new Date()  |      true
}
```

No bloco `expect:` Daniel escreve uma expressão que deve retornar verdadeiro ou falso (que é o único tipo de expressão válida aí) que, neste caso, verifica se um valor diferente de nulo será retornado pela função `incluirCotacao`.

A mágica ocorre no bloco `where:`, que receberá como valor uma tabela delimitada pelo caractere `|`. A primeira linha define os nomes das variáveis e parâmetros usados na expressão escrita em `expect:`. As linhas seguintes apenas referenciam os atributos que definimos no início da nossa classe de acordo com os exemplos que Guto levantou.

Se novas situações surgirem para este caso, basta que incluamos as linhas necessárias na coluna que definimos no bloco `where:`. O resultado? Os testes passaram e Guto saiu bastante feliz com o fato de ter podido contribuir com a evolução do *ConCot*. E mais ainda, porque pôde levar o código-fonte da especificação para uma reunião com a diretoria da *DDL* e todos adoraram saber que agora podem contar com este tipo de recurso a ser aplicado na evolução de todos os sistemas da empresa!

Nesta evolução vimos algumas coisas bastante importantes:

- Como aplicar BDD (ou TDD) não é tão difícil quanto parece.
- Aprendemos a usar o Spock (sim, vimos quase tudo a seu respeito aqui).
- Como escrever testes unitários para serviços.
- Como simular o GORM.

- Como executar testes unitários no Grails.

Daqui para frente, deixaremos a *DDL Engenharia* em paz (por enquanto) para expor os recursos que Grails provê ao desenvolvedor que deseja escrever testes para controladores e bibliotecas de tag. Isso porque você já viu como escrever testes para classes de domínio também (basta usar a anotação `@Mock` e você consegue testar até mesmo suas validações personalizadas).

10.3 Escrevendo testes unitários para controladores

Escrevemos testes unitários para controladores quase exatamente como fizemos para o caso do serviço `ListaCotacoesService`. A diferença está no fato de que agora a anotação `TestFor` receberá como valor uma classe que representa um controlador na nossa aplicação Grails.

Nesta seção, vamos abordar um controlador hipotético bastante simples chamado `TestesController` no qual vamos simular as operações mais comuns que costumamos encontrar na escrita deste tipo de componente.

Na listagem a seguir, podemos ver como ficou o início da nossa especificação `TestesControllerSpec`.

```
package concot

import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(TestesController)
class TestesControllerSpec extends Specification {

}
```

Será criado um novo atributo em nossa especificação chamado `controller`, que usaremos para descrever os comportamentos esperados.

Testando render

Vamos começar pela situação mais simples possível: uma action que renderiza um texto simples. Para tal, imagine que nosso controlador possua uma action chamada `index` cuja implementação vemos a seguir:

```
def index() {
    render "Olá mundo!"
}
```

Como verificaríamos este comportamento?

```
void "action index simples: testando o método render"() {
  when:
   controller.index()
  then:
   response.text == "Olá mundo!"
}
```

É criado um atributo chamado `response` na classe `TestesControllerSpec`, que é uma instância de `GrailsMockHttpServletResponse` que, por sua vez, estende a classe `MockHttpServletResponse` do Spring. Ela contém uma série de métodos úteis que usaremos para testar nossos controladores. Dentre eles, encontra-se `getText()`, que retorna o texto renderizado por uma action.

Em nosso primeiro exemplo, apenas verificamos se o texto retornado quando a action é executada é aquele que esperamos. De um modo geral, todo teste de controladores com Grails é escrito usando apenas os blocos `when:` e `then:`, como no esquema a seguir:

```
def "todo comportamento de controladores é assim"() {
    when: "você apenas invoca a action ou prepara os parâmetros"
        //você tem acesso a todos os escopos
        //usei "params" como exemplo, mas session e flash
        //também estão disponíveis
        params.parametro = "algum valor"
        controller.action()
    then: "e em seguida você verifica se ocorreu o que você espera"

}
```

Verificando se a view correta foi usada

Outra ação comum de um controlador: renderização de uma página GSP. Novamente, algo bastante simples de ser verificado. Imagine a action a seguir:

```
def renderGSP() {
  render view:'renderGSP', model:[nome:"Henrique"]
}
```

Que tal verificar não só se a página correta foi usada, mas também se o `model` é o esperado? Simples!

```
void "testando o model de uma action"() {
   when:
    controller.renderGSP()
   then:"o model deve ter a chave nome"
    model.nome == "Henrique"
    view == "/testes/renderGSP"
}
```

Outro atributo é incluído na classe de especificação: `view`, que armazenará o nome da view usada na renderização. Observe que é usado o caminho completo para este valor. Reparou que é incluído também um atributo `model`? Ele é usado para verificar se o que foi enviado para a página está corretamente preenchido.

Testando um model

Já que falamos de model, que tal expor como testar um? Imagine esta action:

```
def actionModel() {
    [nome:"Henrique", idade:10000]
}
```

Nem precisamos de atributos especiais em nossa classe de especificações neste caso. Basta declararmos a nossa própria variável se quisermos. Afinal de contas, é código Groovy como qualquer outro.

```
def "como verifico um model?"() {
  when:
   modelo = controller.actionModel()
  then:
   modelo.nome == "Henrique"
   modelo.idade > 9000
}
```

Testando redirecionamentos

Talvez nosso controlador possua uma action que selecione o que o usuário deve acessar com base no parâmetro idade, como no código a seguir:

```
def filtroIdade(int idade) {
    if (idade > 18) {
        redirect(action:"entrada")
    } else {
        redirect(uri:"http://www.disney.com")
    }
}
```

Como testamos isso?

```
def "usuário maior de idade"() {
    when:
      controller.filtroIdade(21)
    then:
      response.redirectedUrl == '/testes/entrada'
}

def "usuário menor de idade"() {
    when:
      controller.filtroIdade(17)
    then:
      response.redirectedUrl == 'http://www.disney.com'
}
```

Verificando respostas nos formatos JSON e XML

Não são raros os momentos nos quais você irá usar Grails para criar APIs

ou microsserviços. Nesses casos a escrita de testes é fundamental e, claro, Grails não iria nos deixar na mão aqui.

Sendo assim, vamos a alguns exemplos nos quais vamos especificar o funcionamento dos nossos controladores, primeiro para o formato `JSON`. Imagine que tenhamos uma action como a seguinte:

```
def renderizeJSON() {
  render(contentType:"application/json") {
      [
        titulo:"Falando de Grails",
        ano:2015,
        capitulos:["Introdução", "Groovy: uma breve introdução",
                   "Mergulhando em Groovy", "Precisamos falar sobre Grails
                   "Domínio e persistência", "Buscas",
                   "A camada web: controladores", "A camada web: visualiza
                   "A camada de negócios: serviços", "Testes",
                   "Plugins", "Implantação"]
      ]
  }
}
```

Cuja saída será similar a:

```
{titulo:"Falando de Grails",
 ano: 2015,
 capitulos:["Introdução", "Groovy: uma breve introdução",
                   "Mergulhando em Groovy", "Precisamos falar sobre Grails
                   "Domínio e persistência", "Buscas",
                   "A camada web: controladores", "A camada web: visualiza
                   "A camada de negócios: serviços", "Testes",
                   "Plugins", "Implantação"]}
```

O objeto `response` nos auxilia aqui com a função `getJson` (ou apenas `json` se executado a partir do Groovy) que nos fornece um objeto já parseado com base nos valores `JSON`. Veja como é simples:

```
void "verificando o retorno em JSON. Será bom o livro?"() {
when:
 controller.renderizeJSON()
```

```
then:
 response.json.titulo == "Falando de Grails"
 response.json.ano == 2015
 // Verificando uma lista? Simples!
 response.json.capitulos.length() > 0
}
```

Temos um procedimento bastante similar ao lidarmos com actions que renderizem XML. Vamos a mais um exemplo. Suponha a action a seguir:

```
def renderizeXML() {
 def capitulos = ["Introdução", "Groovy: uma breve introdução",
                  "Mergulhando em Groovy", "Precisamos falar sobre Grails
                  "Domínio e persistência", "Buscas",
                  "A camada web: controladores", "A camada web: visualiza
                  "A camada de negócios: serviços", "Testes",
                  "Plugins", "Implantação"]
 render(contentType:"text/xml") {
    livro(titulo:"Falando de Grails", ano:2015) {
      indice() {
        for (capitulo in capitulos) {
           cap(titulo:capitulo)
        }
      }
    }
 }
}
```

A saída desta action será o XML:

```
<livro titulo="Falando de Grails" ano="2015">
  <indice>
      <cap titulo="Introdução"/>
      <cap titulo="Groovy: uma breve introdução"/>
      <cap titulo="Mergulhando em Groovy"/>
      <cap titulo="Precisamos falar sobre Grails"/>
      <cap titulo="Domínio e persistência"/>
      <cap titulo="Buscas"/>
      <cap titulo="A camada web: controladores"/>
      <cap titulo="A camada web: visualização"/>
```

```
      <cap titulo="A camada de negócios: serviços"/>
      <cap titulo="Testes"/>
      <cap titulo="Plugins"/>
      <cap titulo="Implantação"/>
  </indice>
</livro>
```

Como seria o nosso teste?

```
void "testando o retorno em XML"() {
when:
  controller.renderizeXML()
then:
  response.xml.@titulo.text() == "Falando de Grails"
  response.xml.@ano == 2015
  // listando com a lista de itens no índice?
  response.xml.indice.list().size() > 0
}
```

Assim como no caso dos testes envolvendo `JSON`, aqui há um novo atributo, `xml`, que nos retornará o conteúdo retornado inteiramente parseado. Há pequenas diferenças neste caso. Primeiro, para referenciar atributos, você precisará usar o operador `@`. Segundo, para o acesso a lista de tags você precisará usar a função `list()` tal como exposto no exemplo.

10.4 Testando bibliotecas de tag

Quando descobri que era possível testar *tag libraries* com Grails anos atrás confesso que minha primeira sensação foi choque: até aquele momento, o desenvolvimento deste tipo de artefato era essencialmente interativo; eu apenas modificava o código-fonte e em seguida verificava visualmente se me atendia. Com Grails, pela primeira vez comecei a mudar minha visão sobre este modo de trabalho.

E sabem o que é mais legal? Essencialmente há uma única maneira que irá atender todas as suas necessidades. Sendo assim, tal como fiz no caso dos controladores, vamos implementar uma biblioteca de tag bem simplória apenas para ilustrar os conceitos. Comecemos pela criação da nossa biblioteca

usando o comando `create-tag-lib Testes`, que irá gerar para nós tanto o esqueleto da nossa biblioteca quanto da sua especificação Spock.

O início da nossa *spec* você já pode imaginar como é:

```
package concot

import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(TestesTagLib)
class TestesTagLibSpec extends Specification {

}
```

Será incluído no momento da execução dos testes um novo atributo em `TestesTagLibSpec` chamado `tagLib`, que usaremos para escrever nossos comportamentos. Comecemos pela tag mais simples e inútil possível: uma que sempre imprima o mesmo conteúdo.

```
def apenasUmTexto = {atributos, corpo ->
  out << "Teste!"
}
```

Toda tag ao ser exeutada na realidade retorna um objeto do tipo `org.codehaus.groovy.grails.web.util.StreamCharBuffer`. O que realmente nos interessa é transformá-lo em texto para ver o que obtivemos como resultado. *Como faço isso?* Basta chamar a função `toString()` tal como fizemos em nossa primeira descrição de comportamento.

```
def "quando é um texto simples..."() {
 when:
  // reparou no toString()?
  def resultado = tagLib.apenasUmTexto().toString()
 then:
  resultado == "Teste!"
}
```

Se quiser, também pode usar a função `assertOutputEquals`, em que só precisamos do bloco `expect:`.

```
def "e como uso assertOutputEquals?"() {
 expect:
  /*  Basta apenas digitar como a tag seria
      usada no seu código GSP. Simples, não?  */
  assertOutputEquals "Teste!", "<g:apenasUmTexto/>"
}
```

Mas sabe, muitas vezes você quer apenas verificar se um pedaço bate. Não seria legal poder usar uma expressão regular? Você pode, basta usar a função `assertOutputMatches`!

```
def "e se eu quiser uma expressão regular?"() {
 expect:
  // Atenção para o uso dos parênteses para assertOutputMatches
  // eles são necessários ao lidarmos com expressões regulares
  assertOutputMatches(/.*este.*/, "<g:apenasUmTexto/>"
}
```

10.5 Testes integrados

É fundamental que você se lembre de algumas diretrizes bem simples ao escrever seus testes unitários em Grails. A primeira delas é se lembrar **por que** você os está escrevendo. São uma ferramenta de design e verificação que lhe permite focar sua atenção apenas naquela funcionalidade com que está lidando **naquele** momento.

Se seu teste unitário envolve mais de uma classe, e as ferramentas de mock oferecidas pelo framework não lhe atendem, repense duas coisas: será que seu código não está mais complicado do que deveria e você deveria repensar seu design? E a segunda é: será que você não deveria estar escrevendo testes de integração?

Nesta seção, finalmente veremos como trabalhar com testes integrados. A boa notícia é que o que vimos sobre testes integrados se aplica perfeitamente a este tipo de testes, mas com uma diferença: você não precisa criar mocks, pois todo o sistema estará disponível e tudo o que é necessário fazer é injetar aquilo cujo comportamento queremos projetar ou verificar!

Vamos aqui reescrever a especificação que Daniel criou para `ListaCotacoesService`. Você pode criar testes usando o comando

`create-integration-test` do Grails, que criará um arquivo na pasta `test/integration` seguindo o nome do pacote e o nome do teste a ser criado. Em nosso caso, o conteúdo dos nossos testes de integração em seu estado inicial pode ser visto a seguir:

```
package concot

import grails.test.spock.IntegrationSpec

class ListaCotacoesServiceIntegrationSpec extends IntegrationSpec {

    def setup() {
    }

    def cleanup() {
    }

    void "test something"() {
    }
}
```

Há poucas mudanças: nossa classe agora estende `IntegrationSpec` e não mais `Specification`, porém no momento em que estiver escrevendo suas especificações você não precisa se preocupar com isto. Basta agir quase exatamente como faria no caso dos testes unitários.

Outra diferença importante é que não usaremos mais as anotações `@Mock` e `TestFor`, pois não precisamos simular nenhum componente do sistema: a aplicação estará inteiramente disponível para nós agora. Os métodos `setup` e `cleanup` também funcionam exatamente da mesma maneira. A diferença é que agora vamos interagir com elementos reais da nossa arquitetura como, por exemplo, seu SGBD.

Como não iremos simular objeto algum, podemos injetar o serviço `ListaCotacoesService` em nosso teste integrado:

```
import grails.test.spock.IntegrationSpec
import spock.lang.Shared

class ListaCotacoesServiceIntegrationSpec extends IntegrationSpec {
```

```
    def listaCotacoesService
```

De resto, basta escrever seus testes tal como foi feito no caso dos testes unitários. A grande diferença estará no fato de que agora você não irá referenciar `service`, mas sim `listaCotacoesService`, e neste momento você já sabe fazer isso. ;)

É importante salientar outros comportamentos adotados por Grails na execução dos testes integrados. O primeiro deles diz respeito ao fato de que cada teste é executado em sua própria transação. Finalizada sua execução, é feito o rollback da transação e, com isso, evita-se que seja gerado lixo no seu banco de dados de testes.

Se quiser desabilitar esse comportamento transacional, basta incluir o atributo estático `transactional` com valor igual a `false`, exatamente como faríamos se estivéssemos escrevendo um serviço. Apenas para relembrar, observe como poderíamos desabilitar a natureza transacional dos testes integrados a seguir:

```
import grails.test.spock.IntegrationSpec
import spock.lang.Shared

class ListaCotacoesServiceIntegrationSpec extends IntegrationSpec {

    static transactional = false

}
```

Coisas que você deve levar em consideração em seus testes de integração

O primeiro ponto a ser levado na escrita de testes integrados é o fato de mocks serem completamente desnecessários. Caso se pegue em uma situação na qual se veja forçado a criar um mock, repense: deveríamos realmente estar escrevendo um teste integrado?

Uma boa prática a ser levada em consideração na execução de testes integrados é configurar o `data source` para o ambiente de testes para que o banco de dados seja criado ao iniciarmos a bateria de testes e destruído ao

seu final. Isso evita que sua equipe de desenvolvimento “vicie” o banco de dados, o que terminaria por transferir o clássico problema do “na minha máquina funciona” para o “no meu ambiente de testes funciona”. Fundamental é funcionar no ambiente de produção!

Capítulo 11

Modularizando com plug-ins

Grails nos fornece uma imensa gama de recursos como pôde ser visto durante este livro. Se o leitor mais curioso um dia quiser entender a fundo o framework lendo seu código-fonte, observará algo bastante interessante: o núcleo do Grails é relativamente pequeno e praticamente todas as funcionalidades, como o GORM e scaffolding, por exemplo, são na realidade plug-ins.

Na verdade, Grails é mais que um framework para desenvolvimento web: trata-se de uma plataforma extensível que permite a qualquer um enriquecê-lo com novas funcionalidades através da escrita de plug-ins. Neste capítulo, iremos criar um plug-in bastante simples para o projeto *ConCot*, o que lhe fornecerá uma visão bastante interessante sobre o modo como este importante elemento da arquitetura Grails funciona.

11.1 O que é um plug-in?

Como sempre, partiremos a partir da definição do termo chave do capítulo que é a palavra *plug-in*. Esta se refere a um tipo específico de software, aquele que provê novas funcionalidades a outro preexistente sem que tenhamos de alterar seu código-fonte. Trata-se de um ponto de extensão provido por um *estilo arquitetural*.

Esse *estilo arquitetural* é o que permite a extensão do software e é muito importante que você o compreenda no contexto do Grails. Ao falarmos de uma arquitetura extensível, vêm à mente alguns exemplos bastante conhecidos como, por exemplo, os navegadores que nos permitem executar diferentes conteúdos a partir da instalação de plug-ins, como o Flash ou Java, ou mesmo soluções mais sofisticadas como OSGi [1] ou JBoss Modules [18].

Cada uma destas soluções/arquiteturas possui alguns atributos em comum:

- A aplicação hospedeira, que permite a instalação de plug-ins através do provimento de pontos de extensão.
- Os plug-ins e o modo como eles são implementados pelos desenvolvedores.
- O tempo no qual plug-ins são carregados: compilação, execução?

Nossa aplicação hospedeira é o próprio Grails (que na realidade se chama *Grails Core*). É ele é que fornecerá as APIs e pontos de extensão que serão usados na implementação dos plug-ins.

Como os plug-ins são implementados? Uma surpresa agradável para você: **quase** exatamente igual ao modo como escrevemos aplicações Grails que vimos durante todo este livro. O código-fonte do plug-in possui a mesma estrutura de arquivos e diretórios com que você já está habituado a trabalhar em seus projetos Grails, o que nos trás algumas vantagens:

- Você é capaz de dar manutenção em plug-ins preexistentes pois já sabe como seu código-fonte é estruturado.

- Você e sua equipe são capazes de fazer análises mais detalhadas a respeito de um plug-in que vocês estejam pensando em adotar em seus projetos. Com isso, o número de "surpresas desagradáveis" decorrentes da aplicação de plug-ins mal codificados ou que não atendam suas necessidades diminui bastante. Pense nas bibliotecas normalmente adotadas em nossos projetos Java: quanto tempo você levaria para entender o modo como seu código-fonte é estruturado?

- Já que sua equipe é capaz de entender de forma quase imediata o código-fonte destes componentes, é possível tirar proveito disto para ser sua porta de entrada ao mundo open source provendo melhorias a plug-ins preexistentes (e no código-fonte do próprio Grails) ou mesmo provendo seus próprios produtos!

Quando plugins são carregados? Em tempo de compilação. Você não pode carregar novos plug-ins em uma aplicação Grails após esta ter sido implantada em produção.

Quando você pode tirar proveito dos plug-ins

O termo chave ao pensarmos em plug-ins é *componentização* : você cria um componente quando observa a necessidade de se ter determinada funcionalidade (ou conjunto de) em mais de um projeto e quer evitar a repetição do seu esforço.

Armadilha: componentização precoce

Fig. 11.1: Atenção redobrada!

Somente comece a pensar na criação de um componente após ter observado sua necessidade em dois ou mais casos. Pior que o retrabalho é a componentização precoce ou excessiva, que pode tornar a manutenção dos seus projetos muito mais complexa.

Talvez mais de um dos seus projetos necessitam de funcionalidades que não se encontrem presentes no Grails como, por exemplo, interação com servidores JMS ou agendamento de tarefas tal como provido pelo plug-in Quartz [28]. Este é o uso mais intuitivo de plug-ins: prover novas funcionalidades ao framework, mas não é a única. Aqui seguem algumas possibilidades igualmente interessantes:

A primeira delas é reaproveitamento de código entre seus projetos. Se você optou pelo Grails como framework, é natural que o aplique mais de uma vez entre suas aplicações. Não é raro vermos as mesmas classes de domínio (ou outros artefatos) se repetirem entre diferentes projetos. Pense nas classes que você cria para, por exemplo, lidar com a autenticação e autorização de usuários. Por que repetir este tarefa toda vez ou, ainda pior, ficar copiando arquivos de um ponto para outro? Escreva um plug-in e reaproveite esse conhecimento!

Talvez você possua recursos estáticos ou layouts que deseje reaproveitar em seus projetos e com isso padronizar sua identidade visual na sua empresa. Você pode incluir todos em um mesmo plug-in reaproveitado-os entre suas aplicações. Implemente uma vez seus layouts, reaproveite-os sempre que precisar.

Você pode também ter implementados scripts Gant (Grails 2.x e 1.x) ou Gradle (Grails 3.x pra frente) que automatizem e padronizem o modo como sua equipe trabalha e deseja que sejam adotados em todos os seus projetos Grails.

Outro uso interessante é fazer dinheiro com plug-ins. Não é raro encontrarmos soluções geniais que facilitam imensamente a vida de desenvolvedores pelo mundo. Mais à frente quando formos falar sobre empacotamento de plug-ins, mostrarei como empacotá-los em arquivos binários disponibilizando apenas o conteúdo binário. Já pensou em vender seus plug-ins ou mesmo disponibilizá-los para a comunidade, ganhando com isto uma bela visibilidade ou dinheiro? Apenas uma ideia. ;)

Essas são apenas algumas possibilidades: quando falarmos sobre modularização, sua imaginação é o limite, no entanto o mais importante e que se manterá em todas as oportunidades é bastante simples: ao reaproveitar código você ganha tempo, economiza dinheiro e constrói uma base sólida que poderá ser seu grande diferencial no mercado. Tenha isso em mente.

11.2 Criando nosso primeiro plug-in

Conforme o *ConCot* foi evoluindo e amadurecendo na *DDL Engenharia* a diretoria começou a se interessar cada vez mais pelas possibilidades que Guto alardeava. Tanto que um dia a presidente da companhia, Dedé, resolveu fazer uma visita aos programadores da empresa para se interar mais a respeito. Foi quando ela se encontrou com Daniel, que acabava de voltar do café.

E aí Daniel, tudo bem? Vim aqui dizer um "oi" pois estou muito feliz com os avanços que você junto com Guto e Kico tiveram no *ConCot*. O pessoal do departamento de suprimentos está gostando muito do resultado.

Muito bom ouvir isso Dedé, mas você veio aqui só pra me elogiar???

Sempre desconfiado hein, Daniel? Na realidade vim porque tive uma ideia, sabe... Gosto muito do cadastro de itens do *ConCot*, aquele no qual vocês fornecem a descrição e categorizam os itens cotados pelo sistema. Estava pensando se não seria possível você **copiar** este pedaço para um novo sistema aqui da empresa.

Como assim?

Seguinte: no *ConCot* a gente apenas cota os itens, ou seja, a gente apenas fornece os preços que encontramos, certo?

Sim.

Agora eu preciso de algo novo, que nos ajude a preencher umas planilhas que fazemos o tempo inteiro aqui na empresa. Nestas planilhas incluímos a quantidade de cada equipamento que adotamos em nossos projetos. Por exemplo: *iremos usar 4 britadores do tipo X, 6 motores do tipo Y etc.*, entende?

Hmm... entendo.

Mais do que isto, eu também gostaria que você adicionasse mais campos ao `Item` que não estão relacionadas tanto às cotações quanto aos quantitativos. Queria melhorar um pouco mais esta descrição com estes atributos.

Quais atributos? (Daniel começa a sentir o perigo)

Alguns itens podem não ser mais fabricados, sabe? Então queria um campo chamado "obsoleto" nos itens, assim a gente evitaria a cotação ou inclusão de itens em planilhas que não sejam mais fabricados. E também seria muito legal se a gente pudesse incluir observações nos descritivos dos itens. Você faz isso pra mim?

Mas aí eu vou ter de mexer nos DOIS sistemas!!!!!!

Qual é, cara!!! Você é inteligente e não irá me decepcionar, certo Daniel?

Claro que não! (Daniel sente a pressão)

A criação de um plug-in irá tornar a vida de Daniel muito mais simples e lhe economizará bastante tempo, além de lhe fornecer um ativo reaproveitável que poderá ser usado em futuros projetos da empresa (acredite, eles surgirão).

É fundamental saber **o que** deverá ser incluído em nosso plug-in. Como estamos falando de ativos **reaproveitáveis** e **compartilháveis**, não estamos falando dos nossos controladores. Afinal de contas, há sempre regras de navegação (ao menos no caso da *DDL Engenharia*) específicas para cada aplicação. Também não há regras de negócio implementadas em serviços referentes aos cadastros de itens ou mesmo páginas até agora.

Neste momento, só conseguimos ver como ativo reaproveitável as classes de domínio. E dado que precisaremos inclusive incluir novos campos que serão compartilhados entre o *ConCot* e este novo sistema, elas acabam se tornando nosso alvo a ser compartilhado.

Criando o plug-in

O modo como criamos um plug-in é muito parecido com o que adotamos ao iniciarmos um projeto Grails tradicional. A diferença está no comando: `create-plugin` em vez de `create-app`. É uma excelente prática padronizar os nomes dos seus plug-ins usando algum prefixo que os identifique como pertencentes a algum grupo (como sua empresa, plataforma ou qualquer outro agrupador que lhe faça sentido). Como nosso contexto é a *DDL Engenharia*, nada mais natural que adotemos o prefixo `ddl-plugin` e nosso primeiro plug-in se chamará portanto `ddl-plugin-itens`. Basta executar o comando a seguir:

```
grails create-plugin ddl-plugin-itens
```

Será criado um diretório com o mesmo nome do seu plug-in, exatamente como ocorre quando criamos uma aplicação Grails tradicional. E navegando por esta árvore de diretórios você se sentirá em casa. Por quê? Veja a o conteúdo da pasta criada.

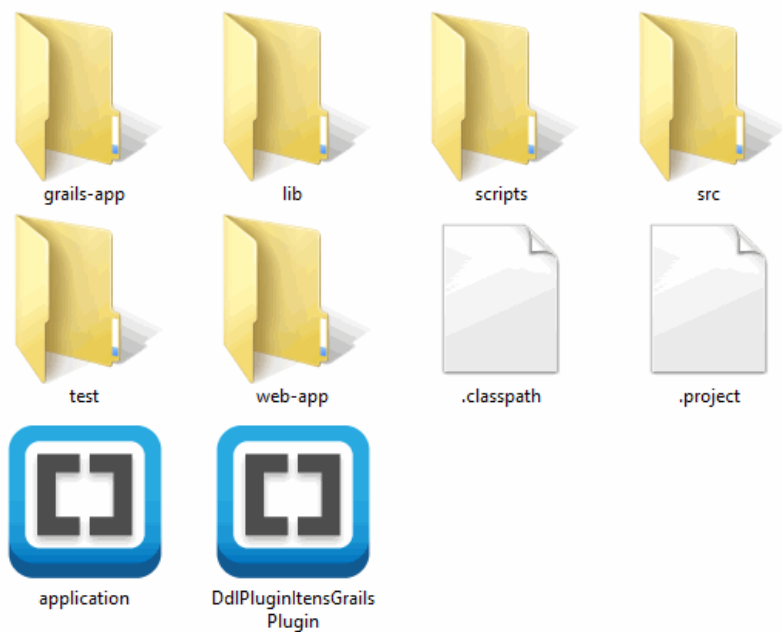



Fig. 11.2: Estrutura de diretórios do plug-in

E qual o conteúdo da pasta `grails-app`?

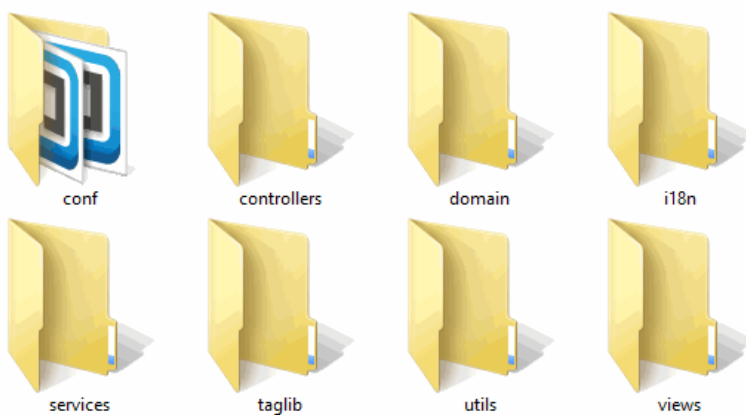



Fig. 11.3: Estrutura de diretórios do plug-in

Exatamente o conteúdo que você esperaria se estivesse criando uma aplicação Grails padrão. E por quê? Porque plug-ins Grails **são** aplicações tradicionais! O conteúdo que você incluirá no projeto do plug-in é quase que exatamente igual ao de uma aplicação convencional, ou seja, se você leu este livro sequencialmente até chegar a este capítulo, já sabe **pelo menos 80%** do que precisa para começar a criar seus próprios plug-ins!

Mudanças na estrutura de diretórios no Grails 3

Fig. 11.4: Compatibilidade

Há pequenas mudanças na estrutura de diretórios a partir da versão 3 do Grails conforme descrevemos no último capítulo deste livro, mas não se assuste, são pequenas alterações apenas.

A única diferença na estrutura de diretórios é a presença de um arquivo que se encontra na raiz do projeto cujo nome é `DdlPluginItensGrailsPlugin.groovy`. Grails sempre irá criar um arquivo chamado `[nome do seu plug-in]Plugin.groovy` no qual iremos incluir diversas metainformações sobre nosso plug-in além de também incluir pontos de extensão bastante interessantes. Observe que o prefixo aplicado ao nome deste arquivo consistirá no nome do seu plug-in no formato *camel cased* removendo o caractere `'-'`.

Identificando o plug-in

Criado o plug-in, o arquivo `DdlPluginItensGrailsPlugin.groovy` é muito parecido ao que expomos a seguir:

```
class DdlPluginItensGrailsPlugin {
    // the plugin version
    def version = "0.1"
    // the version or versions of Grails the plugin is designed for
    def grailsVersion = "2.4 > *"
    // resources that are excluded from plugin packaging
    def pluginExcludes = [
        "grails-app/views/error.gsp"
```

```
    ]

    // TODO Fill in these fields
    def title = "Ddl Plugin Itens Plugin" // Headline display name of the
    def author = "Your name"
    def authorEmail = ""
    def description = '''\
Brief summary/description of the plugin.
'''

    // URL to the plugin's documentation
    def documentation = "http://grails.org/plugin/ddl-plugin-itens"

    // Restante omitido

}
```

Para começarmos nosso desenvolvimento, primeiro precisamos alterar o valor de alguns atributos da classe `DDlPluginIntensGrailsPlugin`. Vamos às chaves principais:

- `version` O número de versão do plug-in. Em nosso caso mudamos para `1.0.0`.
- `grailsVersion` Com quais versões do Grails nosso plug-in é compatível. No nosso caso, manteremos o mesmo valor, pois nosso plug-in foi feito pensado no Grails 2.4, no entanto você poderia escrever algo como `1.1 > *`, que representa "qualquer versão do Grails a partir da 1.1 em diante".
- `title` O nome amigável do plug-in. Em nosso exemplo será `"Cadastro de itens da DDL Engenharia"`. Este nome é o que será exposto, inclusive, como título na página do plug-in no site oficial do Grails caso seja publicado lá.
- `author` O nome do autor (ou autores) deste código-fonte.
- `authorEmail` O e-mail de contato com o autor.

- `description` Texto descritivo curto sobre o plug-in.

Adicionando conteúdo ao plug-in

Preenchidos os metadados necessários, o próximo passo é dar massa ao plug-in. No caso do *ConCot*, tudo o que precisamos fazer é mover algumas de suas classes de domínio para a pasta `grails-app` domain do projeto `DDL Plugin Itens`. Como só iremos lidar com o cadastro de itens, as classes que iremos mover são `Item` e `Categoria` do pacote `concot`.

Para facilitar nossa vida e para fins didáticos apenas, vamos manter a mesma estrutura de pacotes no código-fonte do plug-in. Em casos reais, você irá usar um nome de pacote customizado para o nome do plug-in.

Agora tudo o que precisamos fazer é incluir os campos que Dedé pediu a Daniel que fosse incluídos na classe `Item`. O resultado podemos ver na listagem a seguir:

```
package concot

class Item {

    String nome
    byte[] imagem
    boolean obsoleto = false
    String observacoes

    String toString() {
        this.nome
    }

    static belongsTo = [categoria:Categoria]

    static constraints = {
        nome nullable:false, blank:false, maxSize:128
        categoria nullable:false
        imagem nullable:true, maxSize:65536
        obsoleto nullable:false
        observacoes nullable:true, blank:true, maxSize:1024
    }
```

```
}
```

Por mais incrível que possa parecer, nosso plug-in está pronto. Não precisaremos modificar mais nenhum código-fonte ou arquivo de configuração. Quer dizer, quase pronto, pois ainda precisamos usá-lo.

11.3 Plug-ins inline

Movidas as classes `Item` e `Categoria` do projeto *ConCot*, você não conseguirá mais executá-lo ou empacotá-lo como um `WAR` pois os controladores apontarão para classes que não estarão mais disponíveis no classpath do projeto. Você pode resolver isto de uma forma bastante simples referenciando o código-fonte do plug-in `DDL Plugin Itens`.

Para tal, basta que o código-fonte do plug-in seja acessível ao do seu projeto. Você precisará apenas modificar o arquivo `BuildConfig.groovy` do projeto *ConCot* tal como no exemplo a seguir:

```
// Referência ao nosso plug-in
// Repare que incluímos o nome do plug-in como uma string
// ao final da expressão
grails.plugin.location."ddl-plugin-itens" = "../ddl-plugin-itens"

grails.project.dependency.resolution = {
// restante omitido
```

É muito importante que a instrução `grails.plugin.location.[nome do seu plug-in]` seja incluída **fora** do bloco `grails.project.dependency.resolution`, pois estamos lidando com outro tipo de resolução de dependências aqui: estamos referenciando diretamente o código-fonte do plug-in. Quando executamos comandos como `run-app`, `test-app` ou qualquer outro, o Grails irá, por trás dos panos, mesclar as duas bases de código: a do plug-in e a da aplicação.

Usar plug-ins de forma inline como fizemos aqui é uma excelente prática quando você estiver escrevendo seus próprios plug-ins, pois com isto poderá testar seu funcionamento exatamente como o faria com uma aplicação Grails padrão. Você pode, por exemplo, alterar o código-fonte do plug-in e ver o resultado no seu projeto em execução com o comando `run-app`.

11.4 Empacotando o plug-in

Na maior parte das vezes não usamos plug-ins inline, mas sim providos por terceiros. No caso da *DDL Engenharia*, acredito que no futuro irão adotar este mesmo tipo de comportamento assim que estabilizarem as mudanças necessárias nas classes `Categoria` e `Item`.

No caso de plug-ins, em vez de gerarmos um arquivo `WAR` convencional, vamos gerar por padrão um arquivo com a extensão `zip` contendo todo o código-fonte do nosso plug-in usando o comando `grails package-plugin` que, no caso do *DDL Plugin Itens* irá gerar um arquivo chamado `grails-ddl-plugin-itens-1.0.0.zip` na raiz do projeto.

Outra alternativa interessante é o chamado *plug-in binário*, introduzido com a versão 2.0 do Grails. A principal diferença está no fato de o código-fonte não ser incluído no artefato final, mas sim apenas sua versão compilada. Também não é gerado um arquivo `zip`, mas sim um arquivo `JAR` tradicional. Esta abordagem traz consigo algumas vantagens:

- Você pode publicar seu plug-in em repositórios Maven tradicionais.
- Você pode declará-los como qualquer outra biblioteca Java.
- Dado que o código-fonte não é incluído no artefato final, isso facilita a sua comercialização em casos nos quais não seja interessante disponibilizar esta informação (o código-fonte).
- IDEs conseguem analisar melhor as classes que compõem seu plug-in.

Para gerar um plug-in binário, você só precisa executar o comando `package-plugin --binary`, e será gerado no diretório target do projeto um arquivo com a extensão `jar`. No caso do nosso plug-in de exemplo, este arquivo se chamará `grails-ddl-plugin-itens-1.0.0.jar`.

Há uma vantagem oculta na geração de plug-ins binários: dado que eles são compilados antes de serem empacotados, você conseguirá detectar erros em seu código-fonte que impediriam sua compilação no momento em que o plug-in fosse usado pelo projeto cliente.

11.5 Usando plug-ins: o arquivo BuildConfig.groovy

A última vez que vimos o arquivo `BuildConfig.groovy` foi ao falarmos sobre a configuração do driver JDBC no capítulo em que abordamos o tema persistência 5. Chegou a hora de abordar com detalhes aquela que possivelmente é sua seção mais usada: `grails.project.dependency.resolution`.

É neste bloco que incluímos todas as dependências, isso é, bibliotecas de terceiros que usamos em nossos projetos Grails (e plug-ins). Em seu interior, três blocos nos interessam: `repositories`, `dependencies` e `plugins`.

Bloco `repositories`

Neste bloco, declaramos todos os repositórios Maven que usamos em nosso projeto. Caso todas as suas bibliotecas e plug-ins não tenham sido usados pela sua equipe e vocês não possuam um gerenciador de repositórios interno, tudo o que você deve fazer é deixá-lo inalterado.

Por padrão, serão incluídos os repositórios públicos Maven Central, Grails Central (que inclui todas as bibliotecas terceirizadas usadas pelo framework), Grails Plugins (onde fica o repositório oficial de plug-ins do projeto Grails) e Grails Home (que contém os plug-ins principais usados pelo Grails Core).

Caso seja necessário incluir repositórios terceirizados, tudo o que você precisa fazer é incluir uma chamada à função `mavenRepo`, passando como valor a URL do repositório a ser incluído. No exemplo a seguir podemos ver a customização deste bloco:

```
repositories {
  grailsPlugins()
  grailsHome()
  mavenLocal()
  grailsCentral()
  mavenCentral()
  // Adicionando mais um repositório
  mavenRepo "http://repository.codehaus.org"
}
```

Bloco `dependencies`

Neste bloco vamos declarar todas as dependências externas necessárias para a execução do nosso projeto que não sejam plug-ins. A identificação das dependências é feita adotando-se as convenções do Maven que vimos no capítulo sobre persistência 5.4, como no exemplo exposto a seguir:

```
dependencies {
  runtime 'mysql:mysql-connector-java:5.1.24'
  build "org.fusesource.jansi:jansi:1.11"
}
```

Para cada linha declara-se uma dependência com seu relativo escopo que, atualmente, pode vir nos seguintes sabores:

- `build` A dependência é usada apenas durante o processo de build da aplicação, ou seja, quando executamos os comandos disponibilizados pelo CLI do Grails.
- `runtime` A dependência será incluída no artefato final da aplicação, ou seja, estará contida no arquivo `WAR` que iremos gerar.
- `compile` A dependência é necessária durante os momentos de compilação (*compile time*) e `runtime`.
- `test` A dependência estará presente apenas durante a execução dos testes automatizados do projeto. Afinal de contas, para que enviar as bibliotecas do Spock para o seu servidor?
- `provided` A aplicação precisa da dependência em tempo de execução, no entanto ela já é provida pelo servidor de aplicações ou servlet container. Exemplo: a API Servlet.

Bloco `plugins`

É o bloco que realmente nos interessa quando o assunto é o uso de plug-ins. Seu funcionamento é exatamente igual ao do bloco `dependencies`. A diferença está no fato de que aqui são declarados os plug-ins usados pela nossa aplicação.

Normalmente, tudo o que você precisa fazer para usar seus plug-ins é copiar para este bloco a declaração de dependência fornecida pelos próprios autores do plug-in. No caso do Spring Security Core, por exemplo, basta incluir a seguinte instrução:

```
plugins {
  compile ":spring-security-core:2.0-RC4"

  // restante omitido para fins didáticos
}
```

BuildConfig.groovy não existirá no Grails 3

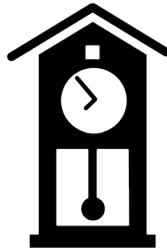



Fig. 11.5: Compatibilidade

A versão 3.0 do Grails substitui o Gant como ferramenta de build padrão do framework pelo Gradle, uma alternativa muito mais poderosa e também inteiramente baseada na linguagem Groovy. Essa é uma mudança bastante profunda no funcionamento interno do framework e, dentre as principais mudanças está o fato de que o arquivo `BuildConfig.groovy` não mais existirá em futuras versões do framework.

A partir da versão 3.0 do framework, todos os plug-ins passarão a ser empacotados por padrão no formato binário, e seu uso pelas aplicações será exatamente como o faríamos com qualquer outra biblioteca ou framework Java. Vamos apenas referenciar sua identificação no padrão Maven.

11.6 O que não entra no plug-in

Tão importante quanto saber o que incluímos em um plug-in é a lista dos artefatos que não são incluídos na distribuição dos nossos plug-ins pelo Grails. É uma lista relativamente pequena:

- `grails-app/conf/BootStrap.groovy`
- `grails-app/conf/BuildConfig.groovy` (apesar de que é usado para incluir no pacote do plug-in um arquivo chamado `dependencies.groovy`, que lista todas as dependências necessárias para a execução do plug-in).
- `grails-app/conf/Config.groovy`
- `grails-app/conf/DataSource.groovy`
- `grails-app/conf/UrlMappings.groovy`
- `grails-app/conf/spring/resources.groovy`
- Tudo o que estiver no diretório `/web-app/WEB-INF`
- Tudo o que estiver no diretório `/web-app/plugins/`
- Tudo o que estiver no diretório `/test/`

Além desta lista padrão, você também pode instruir o Grails a não incluir parte do seu código-fonte na distribuição do seu plug-in. Para tal, basta que no arquivo de descrição do plug-in (em nosso caso, `DdlPluginItensGrailsPlugin.groovy`) seja preenchido o atributo `pluginExcludes` tal como no exemplo a seguir, em que informamos o Grails a não incluir o arquivo `grails-app/views/error.gsp`:

```
def pluginExcludes = [
   "grails-app/views/error.gsp"
]
```

Customizar os arquivos que podem ou não ser incluídos no artefato final do seu plug-in possibilita ao desenvolvedor criar o seu próprio ambiente de desenvolvimento para a geração deste tipo de projeto. A semelhança dos plug-ins com aplicações Grails padrão não está apenas na estrutura de arquivos e diretórios: **todos** os comandos do CLI do Grails funcionam no desenvolvimento de plug-ins.

Sendo assim, você pode, por exemplo, criar controladores e arquivos GSP para simular o comportamento do seu plug-in em outras aplicações e, na hora de gerar o pacote final, simplesmente excluir estes artefatos da geração final.

11.7 Adicionando métodos dinâmicos em tempo de execução

11.7. ADICIONANDO MÉTODOS DINÂMICOS EM TEMPO DE EXECUÇÃO

Escrever plug-ins vai além de simplesmente fornecer classes de domínio, serviço, controladores e recursos estáticos. Já se perguntou como o GORM adiciona todos aqueles métodos em nossas classes de domínio?

No arquivo descritor de plug-ins há um atributo chamado `doWithDynamicMethods`, que recebe como valor uma closure cujo único parâmetro se chama `applicationContext` (que é o contexto do Spring). Mas o que realmente nos interessa é um atributo chamado `grailsApplication`, que nos fornece acesso a todos os artefatos padrão do Grails como, por exemplo, controladores, classes de domínio e serviços. No exemplo a seguir vamos incluir um método que apenas imprime "Olá mundo" em todos os controladores do projeto no qual o plug-in for aplicado.

```
def doWithDynamicMethods = { applicationContext ->
    for (controlador in grailsApplication.controllerClasses) {
        controlador.metaClass.digaOi = {->
            println "Olá mundo"
        }
    }
}
```

O objeto `grailsApplication` é uma implementação da interface

`grails.commons.GrailsApplication`. Para acessar a lista de artefatos pertencentes a determinada categoria, basta usar a seguinte convenção ao buscar por esta propriedade: `<tipo do artefato>Classes`, onde tipo do artefato pode ser:

- domain
- controller
- tagLib
- service
- codec
- bootstrap
- urlMappings

11.8 Entendendo a sobrescrita de artefatos em plug-ins

11.8. ENTENDENDO A SOBRESCRITA DE ARTEFATOS EM PLUG-INS

É muito importante entender como os artefatos presentes em um plug-in são inseridos na aplicação cliente, pois isso lhe possibilita customizar o funcionamento dos seus plug-ins sem modificar seu código-fonte e também evitar problemas que, sem este conhecimento, são bem difíceis de serem detectados e entendidos ao usarmos um plug-in.

Ao executarmos comandos que envolvam o empacotamento de uma aplicação ou sua execução (incluindo execução de testes), o Grails irá copiar o código-fonte dos plug-ins para dentro da estrutura de diretórios da aplicação. No entanto, caso na aplicação exista algum artefato cujo nome completo (o que inclui o diretório no qual encontra-se armazenado) coincida com o nome do artefato presente em algum plug-in, será mantida a versão presente no código-fonte da aplicação.

Sendo assim, voltando ao *ConCot*, imagine que exista o arquivo `grails-app/domain/concot/Item` tanto no código-fonte da aplicação quanto no plug-in que construímos neste capítulo. Será mantido aquele presente no *ConCot*.

11.9 Preparando-se para o Grails 3

11.9. PREPARANDO-SE PARA O GRAILS 3

Enquanto este livro é finalizado, a versão 3.0 do Grails se aproxima. Até este momento usamos apenas a versão 2.4.4 para nossos exemplos, no entanto é importante que você saiba de algumas coisas que irão mudar o modo como lidamos com plug-ins.

Para começar, o modo como os criamos continuará sendo exatamente o mesmo. O comando `create-plugin` ainda existirá, assim como todos os demais. A principal mudança ocorrerá no modo como os empacotamos. O modelo padrão agora será o binário, possibilitando-nos com isto tirar máximo proveito da infraestrutura de repositórios Maven já existente hoje.

O modo como usamos plug-ins também será modificado. Não mais usaremos o arquivo `BuildConfig.groovy` para declarar nossas dependências, mas sim os arquivos de configuração do Gradle. Infelizmente, ainda estamos no release RC-1 do Grails 3.0, e as coisas estão se modificando com muita rapidez, no entanto a próxima atualização deste livro (que será em breve, assim que sair a versão final do Grails 3.0) conterá maiores informações sobre como proceder.

Capítulo 12

Grails 3.0

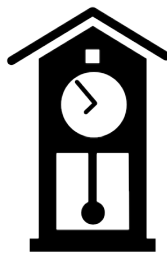



Fig. 12.1: Compatibilidade

A versão 3.0 do Grails saiu enquanto eu terminava este livro. Até aqui trabalhamos com a versão 2.4.4, mas não se assuste: o que aprendemos se manterá

na nova versão e todo o seu aprendizado não será perdido, no entanto é importante que sejam expostas as principais mudanças que esta nova versão nos traz.

Um pouco antes de sair a versão 3.0, saíram as versões 2.4.5, que contém algumas correções de bugs e a 2.5.0. A principal novidade na versão 2.5 foram as versões do Spring e Groovy que foram mudadas para, respectivamente, as versões 4.1 e 2.4. Sendo assim, o que vimos até aqui não muda em quase nada.

12.1 Sai o Gant, entra o Gradle

Desde sua primeira versão, Grails sempre usou como mecanismo de build o Gant, que é uma ferramenta de construção (*build*) bastante poderosa e que nos serviu muito bem por todo este período. No entanto, conforme o tempo foi passando novas opções foram surgindo, dentre elas o Gradle que vai além do Gant: trata-se de uma ferramenta de gerencia de projetos que lida desde a gestão de dependências, padronização de diretórios, ciclo de vida, construção e muito mais. Pense no Gradle como um *Gant com anabolizantes*.

É por esta razão que não falamos sobre o Gant neste livro: este é um conhecimento que você **não levará** para o Grails 3.0. Daqui para frente, será apenas o Gradle, que agora permeia toda a construção do framework. Esta mudança não veio apenas para jogar fora todos os seus velhos scripts de build escritos em Gant. Há vantagens significativas na adoção do Gradle como:

- Melhor suporte das IDEs: será possível, por exemplo, usar seus projetos Grails com a versão gratuita do IntelliJ, que oferece suporte ao Gradle.
- A adoção de uma ferramenta de gestão de ciclo de vida com uma comunidade de usuários **muito** maior e mais ativa.
- Muito mais facilidades na integração do seu projeto Grails com ferramentas de integração contínua como Jenkins, Hudson, TeamCity e muitas outras.
- O Gradle possui uma quantidade **imensa** de plug-ins da qual você poderá tirar proveito em seus projetos.

Nova estrutura de arquivos e diretórios do projeto

Uma das consequências interessantes na adoção do Gradle foi a mudança na estrutura de diretórios de projetos Grails. São mudanças que, se comparadas às versões anteriores do Grails, inclusive fazem bastante sentido. Primeiro vamos à mudança de localização de alguns diretórios:

- `src/groovy` e `src/java` Agora se encontram em `src/main/groovy`. Grails é esperto o suficiente para conseguir distinguir código Java de Groovy. É uma mudança bastante interessante, pois na prática trata-se de um diretório a menos para você se preocupar.
- `test/unit` Agora se encontra em `src/test/groovy`.
- `test/integration` Foi para `src/integration-test/groovy`.
- `web-app` Mudou-se para `src/main/webapp`.

Alguns arquivos também mudaram de "endereço" ou mesmo deixaram de existir, sendo substituídos por alguma outra solução.

- `grails-app/conf/BuildConfig.groovy` Não mais existe, agora todas as configurações de construção encontram-se no arquivo do próprio Gradle, que fica na raiz do seu projeto e se chama `build.gradle`.
- `grails-app/conf/Config.groovy` Agora se chama `application.groovy` e foi renomeado com o objetivo de ficar mais próximo do padrão adotado pelo Spring Boot (falaremos mais sobre ele a seguir).
- `grails-app/conf/UrlMappings.groovy` Uma mudança que fez bastante sentido: agora fica em `grails-app/init`, pois o diretório `grails-app/conf` não é mais considerado um local onde é armazenado código-fonte (o arquivo `Config.groovy` é de configuração).
- `*GrailsPlugin.groovy` Agora ficará em `src/main/groovy`.

Alguns arquivos não existem mais no Grails 3.0, mas nada com o qual você deva se preocupar também. Na prática sua vida acabou ficando um pouco mais simples agora. Segue a lista:

- `application.properties` Na prática só usávamos este arquivo para definir o número de versão do projeto. Dado que agora estamos usando o Gradle, e em seu arquivo de configuração `build.gradle` é possível definir propriedades relativas ao que será construído, por que não usá-lo? E isso foi feito, pois ele foi substituído por `build.gradle`.
- `grails-app/conf/DataSource.groovy` Yeap, não mais existe. Você agora usará apenas o arquivo `grails-app/conf/application.yml` sobre o qual falaremos mais tarde. É uma boa notícia, pois trata-se de menos um arquivo de configuração para nós.
- `web-app/WEB-INF/applicationContext.xml` Este arquivo que era instalado pelo comando `install-templates` foi removido. Agora todos os seus beans customizados deverão ser incluídos em `grails-app/conf/spring/resources.groovy`.
- `web-app/WEB-INF/sitemesh.xml` O filtro do Sitemesh não é mais usado, então o arquivo se torna completamente desnecessário agora.

A pasta `target`, na qual antes era incluído o arquivo `WAR` gerado pelo comando `war` também não existe mais. Foi substituída por outra chamada `build`.

Seu novo arquivo de configuração: `application.yml`

O antigo arquivo `grails-app/conf/Config.groovy` foi substituído por `grails-app/conf/application.yml`, que adota o formato `YAML` (*Yet Another Markup Language*). Este será o formato padrão adotado pelo framework.

No entanto, caso prefira o formato Groovy, basta pegar seu arquivo `grails-app/conf/Config.groovy` e renomeá-lo para `application.groovy` que deverá funcionar sem problemas na maior

parte dos casos. É importante salientar que, neste momento em que apenas temos o Grails 3.0-RC1, não temos ainda certeza se este procedimento funcionará sem qualquer tipo de transtorno para vocês.

Vale relembrar aqui o que foi dito agora há pouco: o conteúdo do arquivo `DataSource.groovy` agora se encontra em sua integridade aqui, sendo assim é menos um arquivo para você se preocupar. Por um lado isso é bom, por outro, trata-se de um arquivo maior agora também.

Não existe mais a pasta `lib`

Um dos diretórios mais úteis quando precisávamos reaproveitar código-fonte legado era este diretório no qual copiávamos nossos arquivos `JAR`. Como você deverá fazer isto agora? Simples, basta usar o arquivo `build.gradle` que lhe parecerá bastante familiar, visto que o arquivo `BuildConfig.groovy` com o tempo foi se aproximando aos poucos deste em seu formato e aparência.

Nele há também um bloco chamado `dependencies` e `repositories`. No caso, para criar sua própria pasta "lib retrô" basta seguir este procedimento.

1) Crie uma pasta chamada `lib` na raiz do seu projeto.

2) Edite o arquivo `build.gradle` para que fique similar ao exemplo a seguir:

```
repositories {
    // Adicione o bloco "flatDir"
    flatDir {
        dirs 'lib'
    }
}
```

Inclusão de plug-ins

Não há mais o arquivo `BuildConfig.groovy` como já disse neste capítulo, sendo assim, basta agora usar o arquivo `build.gradle`. Neste, há um bloco chamado `dependencies`. Novamente, tudo o que você precisa fazer é

declarar seu plug-in como se fosse uma dependência, exatamente como faria antes. A diferença é que não há mais o bloco `plugins`.

12.2 Uma nova base: Spring Boot

Sem sombra de dúvidas, um dos projetos mais importantes e excitantes dentro do ecossistema Spring lançado nos últimos anos foi o *Spring Boot*. Assim como Grails, trata-se de um *metaframework*, construído com base no Spring 4.1, que nos traz uma série de vantagens e facilidades interessantes:

- Possibilita empacotar nossos projetos web como arquivos `JAR`. Nestes arquivos será embutido um servlet container (Tomcat ou Jetty), facilitando bastante o processo de implantação.
- Uma arquitetura de módulos extremamente rica e fácil de usar. Você tem acesso a, por exemplo, módulos de auditoria, segurança e todo o ecossistema Spring acessível de uma forma extremamente simples.

Neste momento, a principal novidade prática para nós é a inclusão de um novo comando chamado `package`, que irá gerar um arquivo `JAR` no interior da pasta `build` do seu projeto. Com ele, você não precisa mais fazer a implantação em um servidor de produção. Basta executar o comando `java -jar [nome do arquivo gerado].jar` e sua aplicação será iniciada.

Por trás dos panos o Grails, ou melhor, o Spring Boot, irá embarcar o Tomcat em sua aplicação, o que poderá facilitar bastante a implantação do seu projeto, visto que não é mais necessário gerar um `WAR` e executar o procedimento de implantação (*deploy*) padrão do seu servidor de aplicação ou container servlet.

Claro, o comando `war` ainda existe e permanecerá caso seja necessário gerar um `WAR`. Quando usar um ou outro? Depende muito do caso. Se você precisa do poder de um servidor de aplicações, `WAR` é o caminho, especialmente quando a configuração do pool de conexões com o banco de dados é feita por ele. No caso do `JAR`, modificar o pool irá requerer a geração de um novo pacote.

12.3 Qual o melhor caminho para o upgrade?

De acordo com a equipe de desenvolvimento do Grails [27], o procedimento mais fácil consiste em executar os passos a seguir.

1) Crie um novo projeto com o Grails 3.0-RC1;

2) Copie seu código-fonte do projeto antigo para o novo;

3) Atualize seus arquivos de configuração.

Infelizmente é um processo manual neste momento e que pode ser um pouco trabalhoso, mas não é uma tarefa tão hercúlea quanto aparenta.

12.4 Finalizando?

Se você chegou ao final deste livro e o leu de cabo a rabo, é sinal de que gostou do que viu aqui. Muito obrigado! Espero que tenha ficado claro que Grails é muito mais que um framework para desenvolvimento web ou uma plataforma: trata-se de uma maneira diferente de vermos e usarmos a plataforma Java EE.

A versão 3.0 é em sua essência uma reescrita quase completa do framework, no entanto é interessante observar que este modo de trabalho se manteve praticamente inalterado. Ainda há as mesmas convenções (apenas alguns arquivos mudaram de lugar), a mesma interface de linha de comando e, o que considero mais importante: trata-se de uma ferramenta que nos faz repensar de uma forma bastante profunda a maneira com a qual lidamos com uma das plataformas mais poderosas da história da computação, que é o Java Enterprise Edition.

Acredito que ainda há uma longuíssima estrada pela frente para este framework, especialmente para as tecnologias baseadas em Groovy que vimos nos primeiros capítulos deste livro. Torço para que se no pior caso possível, ou seja, se você jamais vier a usar o Grails em seus projetos, ao menos ele tenha te forçado a repensar seu modo de trabalho.

Muito obrigado!

Capítulo 13

Apêndice 1 Lista de dialetos do Hibernate

- Caché 2007.1 `org.hibernate.dialect.Cache71Dialect`
- CUBRID 8.3+ `org.hibernate.dialect.CUBRIDDialect`
- Oracle 9 DataDirect Driver `org.hibernate.dialect.DataDirectOracle9`
- DB2/390 `org.hibernate.dialect.DB2390Dialect`
- DB2/400 `org.hibernate.dialect.DB2400Dialect`
- DB2 `org.hibernate.dialect.DB2Dialect`
- Derby 10.5 `org.hibernate.dialect.DerbyTenFiveDialect`

- Derby 10.6 `org.hibernate.dialect.DerbyTenSixDialect`
- Derby 10.7 `org.hibernate.dialect.DerbyTenSevenDialect`
- Firebird `org.hibernate.dialect.FirebirdDialect`
- Frontbase `org.hibernate.dialect.FrontBaseDialect`
- H2 `org.hibernate.dialect.H2Dialect`
- HSQL (HyperSQL) `org.hibernate.dialect.HSQLDialect`
- Informix `org.hibernate.dialect.InformixDialect`
- Ingres 10+ `org.hibernate.dialect.Ingres10Dialect`
- Ingres 9.3+ `org.hibernate.dialect.Intres9Dialect`
- Ingres 9.2 `org.hibernate.dialect.IngresDialect`
- Interbase `org.hibernate.dialect.InterbaseDialect`
- JDataStore `org.hibernate.dialect.JDataStoreDialect`
- McKoi SQL `org.hibernate.dialect.MckoiDialect`
- MySQL 5 `org.hibernate.dialect.MySQL5Dialect`
- MySQL 5.x para o motor de armazenamento InnODB `org.hibernate.dialect.MySQL5InnoDBDialect`
- MySQL até a versão 4.x `org.hibernate.dialect.MySQLDialect`
- MySQL para o motor de armazenamento MyISAM `org.hibernate.dialect.MySQLISAMDialect`
- Oracle 10g `org.hibernate.dialect.Oracle10gDialect`
- Oracle 9i `org.hibernate.dialect.Oracle9iDialect`
- Oracle 8i `org.hibernate.dialect.Oracle8iDialect`
- Pointbase `org.hibernate.dialect.PointbaseDialect`

- Postgres Plus `org.hibernate.dialect.PostgresPlusDialect`
- Postgres 8.1 `org.hibernate.dialect.PostgreSQL81Dialect`
- Postgres 8.2+ `org.hibernate.dialect.PostgreSQL82Dialect`
- Unisys 2200 Relational Database (RDMS) `org.hibernate.dialect.RDMSOS2200Dialect`
- SAP DB `org.hibernate.dialect.SAPDBDialect`
- Microsoft SQL 2005 `org.hibernate.dialect.SQLServer2005Dialect`
- Microsoft SQL Server 2008 `org.hibernate.dialect.SQLServer2008Diale`
- Microsoft SQL Server 2000 `org.hibernate.dialect.SQLServerDialect`
- Sybase 11.9.2 (evita a sintaxe ANSI JOIN) `org.hibernate.dialect.Sybase11Dialect`
- Sybase Anywhere `org.hibernate.dialect.SybaseAnywhereDialect`
- Sybase Adaptive Server Enterprise (ASE) 15.7+ `org.hibernate.dialect.SybaseASE157Dialect`
- Sybase Adaptive Server Enterprise (ASE) 15+ `org.hibernate.dialect.SybaseASE15Dialect`

Referências Bibliográficas

[1] OSGi Alliance. Website oficial da osgi alliance. http://www.osgi.org/.

[2] Peter Pin-Shan Chen. The entity relationship model - toward a unified view of data. 1976.

[3] Equipe de desenvolvimento do Grails. createcriteria. http://grails.org/doc/2.4.4/ref/Domain%20Classes/createCriteria.html, 2014.

[4] Equipe de desenvolvimento do Grails. Dynamic finders. http://grails.org/doc/2.4.4/guide/GORM.html#finders, 2014.

[5] Equipe de desenvolvimento do Grails. Understanding cascading updates and deletes. http://grails.org/doc/2.4.4/guide/GORM.html#cascades, 2014.

[6] Equipe de desenvolvimento do Groovy. Groovy builders. http://groovy.codehaus.org/Builders.

[7] Equipe de desenvolvimento do Hibernate. Identifier generator - documentação do hibernate. http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/mapping.html#mapping-declaration-id-generator.

[8] Equipe de desenvolvimento do Hibernate. Hql: The hibernate query language. 2014.

[9] Equipe de desenvolvimento do Hibernate. Javadoc da classe scrollableresults. http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/ScrollableResults.html, 2014.

[10] Equipe de desenvovlimento do Grails. Upgrade da versão 2.3 para 2.4. http://grails.github.io/grails-doc/2.4.0/guide/upgradingFrom23.html.

[11] Projeto Apache Foundation. Apache tiles. http://tiles.apache.org.

[12] David Heinemeier Hansson. Tdd is dead. long live testing. http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html.

[13] David Heinemeier Hansson. How to build a blog engine in 15 minutes with ruby on rails. http://www.youtube.com/watch?v=Gzj723LkRJY, 2005.

[14] Martin Heidegger. *Ser e Tempo*. 1927.

[15] Graeme Rocher Henrique Lobo Weissmann. Graeme rocher sobre a obsolência das tags ajax. https://groups.google.com/forum/#!topic/grails-dev-discuss/4yesijtFSB4.

[16] Hibernate. Transitive persistence. http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/objectstate.html#objectstate-transitive.

[17] David Hunt, Andrew; Thomas. *The Pragmatic Programmer*. 1999.

[18] Equipe JBoss. Documentação do jboss modules. https://docs.jboss.org/author/display/MODULES/Home.

[19] Luke Daley e Peter N. Steinmetz Marc Palmer. Documentação de referência do resources plugin. http://grails-plugins.github.io/grails-resources/.

[20] Dan North. Behaviour driven development. http://behaviour-driven.org/.

[21] Dan North. Introducing bdd. http://dannorth.net/introducing-bdd/.

[22] Oracle. Default methods. http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html.

[23] Oracle. Javadoc simpledateformat. http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html.

[24] Oracle. Preventable phenomena and transaction isolation levels. http://docs.oracle.com/cd/B12037_01/server.101/b10743/consist.htm#sthref1919.

[25] OWASP. Cross-site scripting (xss). https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29.

[26] R. Salz P. Leach, M Mealing. Rfc 4122 - uuid. http://tools.ietf.org/html/rfc4122.

[27] Graeme Rocher. Upgrade da versão 2.x para 3.0 do grails. http://grails.github.io/grails-doc/latest/guide/upgrading.html.

[28] Ryan Vanderwerf Sergey Nebolsin, Graeme Rocher. Quarts plugin for grails. https://grails.org/plugin/quartz.

[29] StackOverflow. Using hibernate's scrollableresults to slowly read 90 million records. http://stackoverflow.com/questions/2826319/using-hibernates-scrollableresults-to-slowly-read-90-million-records, 2013.

[30] Alan M. Turing. On computable numbers, with an application to the entscheidungs problem. 1936.

[31] Henrique Lobo Weissmann. Vire o jogo com spring framework. http://www.casadocodigo.com.br/products/spring.

[32] Wikipedia. Binary search. http://en.wikipedia.org/wiki/Binary_search_algorithm.

[33] Wikipedia. Lista de cabeçalhos http. http://en.wikipedia.org/wiki/List_of_HTTP_header_fields.